

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF





网络运维自动化资深专家撰写，8位专家联袂推荐，网络工程师转型必备指南

以场景与实践驱动，涵盖NetDevOps理念、常用工具、编程基础、网络运维常用Python模块、网络设备的数据处理等

# NetDevOps 入门与实践

余欣 著



机械工业出版社  
China Machine Press





## 作者简介

---

### 余欣

思科中国资深系统工程师，近20年网络规划设计与运维经验，曾先后就职于瞻博网络、阿里巴巴、京东金融以及博科等公司。有丰富的互联网一线公司实践经验。擅长大规模运营商级网络、大型园区网以及IDC网络的规划设计与实施。拥有CCIE、JNCIE等认证。





# NetDevOps 入门与实践

余欣 著



机械工业出版社  
China Machine Press





## 图书在版编目 (CIP) 数据

NetDevOps 入门与实践 / 余欣著. —北京: 机械工业出版社, 2018.5  
(网络专业人员书库)

ISBN 978-7-111-59909-8

I. N… II. 余… III. Linux 操作系统—程序设计 IV. TP316.85

中国版本图书馆 CIP 数据核字 (2018) 第 086384 号

## NetDevOps 入门与实践

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 高婧雅

责任校对: 殷 虹

印 刷: 三河市宏图印务有限公司

版 次: 2018 年 5 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 21.5

书 号: ISBN 978-7-111-59909-8

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东







## Praise 本书赞誉

(本书推荐人排名不分先后，以音序排列)

随着通信技术近年来颠覆性的发展和变化，对传统网络技术工程师们的挑战越来越大。作者以传统网工<sup>⊖</sup>成功转型的亲身经验撰写了本书，它直击传统网络工程师们的痛点，是难得的兼具实用价值和实践意义的“惊艳”之作，令人耳目一新！

——方芳，思科大中华区副总裁兼运营商 & 媒体广电事业技术部总经理

网络运维和系统运维本不是一个世界。技术栈、操作任务甚至运维价值观都是截然不同的，一直以来泾渭分明，各自精彩。

近年来虚拟网络的发展、SDN 的兴起，网络与 IT 系统逐渐开始跨界融合，而结合部分的故障定位、全局性的问题跟踪和优化成了传统运维的新盲区；云计算规模化的环境下，海量操作变更、复杂的关联定位，对传统人肉运维来说更是不可承受之痛。新的形势下，传统网络运维工程师的自我救赎之路，就是本书所倡导的 NetDevOps 理念：补齐 IT 系统技术栈，掌握必要的开发语言，熟悉主流的批量运维工具和基础服务，将自动化运维的理念延伸到网络领域，将研发的思维模式嵌入到传统的网络运维动作中，将网络运维标准化、自动化、智能化。

本书深入浅出展示了 NetDevOps 的理念、基础知识和最佳实践，值得有意转型的网络工程师深入研究学习。

——林恩华，中国移动苏州研发中心广州支持中心总经理助理

网络运维可视化、自动化和智能化的快速发展背后的本质诉求是能满足大型互联网公司的巨大网络规模增速和高效高质运维要求，具体又体现在人均运维效率和稳定性指标的极致追求上。每一位互联网企业的网络工程师都恰逢其时，有幸在网络运维领域引领技术发展的潮头并对各行各业中网络技术的发展产生一定的影响。网络运维 DevOps 就是网络工程师发展的方向，已在大型互联网公司深深扎根、蓬勃发展。余欣在阿里巴巴工作期间

---

⊖ 业界对网络工程师的简称，下文也会出现。





经历了网络工程师队伍转型的剧痛，并表现出了优秀的 DevOps 思路和能力。这本书作为网络 DevOps 入门指南写得深入浅出，非常符合网络 DevOps 的实际工作，各种细化的小场景、小步骤非常接地气，同时又富含 DevOps 的深层思想，我相信对传统网络工程师或初入行的网络工程师来说深具价值，推荐给大家研读学习。

——刘洋，阿里巴巴网络系统事业部总经理

伴随互联网业务的高速发展，网络规模持续快速增长，数量庞大的网络设备产生海量的运营数据，传统的人机交互的运维方式面临巨大的挑战。NetDevOps 利用 DevOps 的理念，推进网络运维的自动化与智能化，给网络运维带来了转机。本书介绍了 NetDevOps 产生的背景、发展历程，同时系统阐述了 NetDevOps 的框架体系、工具以及基本的软件编程知识，是国内难得的一本专业而又全面讲解 NetDevOps 技术的学习资料和参考手册，相信希望了解 NetDevOps 的网络同行们，能从本书中找到你们想要的内容。

——邵华，腾讯网络平台部网络架构中心总监

在软件工程领域中，DevOps 已经由一种文化演变成广泛落地的业务思维，将组织内的各个角色更紧密地联系在一起以提高生产力。但是在网络工程领域，受限于网络工程师技术栈及运维管理定势，如何理解 NetDevOps 思想进而在实际工作中更好地解决运维管理问题和新技术部署带来的挑战，仍存在不小的困难。

很高兴看到余欣用简明的语言和具体的场景将 NetDevOps 的方法论和实践进行了系统全面的呈现，是网络工程师、网络平台开发工程师不可错过的参考读物。

——宋磊，百度网络运维部技术经理

本书作者是网络行业的资深老兵，在 Cisco、Juniper 这样的网络设备制造商工作多年，也曾阿里巴巴、京东金融的网络部门从事实际运维工作，拥有丰富的经验，亲身经历了 IP 网络的爆发式增长时代。面对最新的网络自动化运维的趋势，大量的传统运维工作必须转向软件自动化的方式，新的 SDN、NFV 等理念，也要求网络工程师具备软件编程能力。很多老网工在新的挑战面前，会有些眼花缭乱，不知从何入手。本书分享了作者自身的转型经验及丰富的实际案例，指出了一条切实可行的转型道路，对广大网工有非常好的参考价值，尤其是没有软件编程基础的网工。本书由浅入深地介绍了基本的概念和常用的工具，可以让大家少走弯路，节省很多自己去摸索试错的时间和精力。

——王卫，原瞻博网络大中国区总裁

由浅入深，有料清晰！作者结合自身在多家国际网络设备制造商和互联网公司的丰富经验，为读者指明了一条从传统向 NetDevOps 发展的转型之路。纯干货！值得一读！

——徐志骏，思科大中国华东区运营商事业部技术总监

本人与作者在 Brocade 共事期间，我们就意识到让老网工们快速转型 SDN 工程师是







不现实的，因为机器对机器的软件接口（API）不是网工们熟知的。找到一条有实战价值，门槛相对合理，容易启动的“工农结合”的路径就显得格外有吸引力。当前，作为一名新一代云网融合服务商的 CTO，团队建设的一个重要挑战和机会就是赋能老网工们，把建设运维实战经验与智慧总结形成清晰套路（算法），与专业码农们紧密配合，迅速实现运维故障经验软件化、自动化。与此同时，给网工们提供现实的发展演进路径，在实战项目中以商业价值目标为导向培养编程思维，接触机器接口，在一个个自动化的小任务中一步步实现自己的想法，获得真实成就感，成为新一代高度软件化的网络工程师、架构师和产品经理。针对这一目标，本书对 NetDevOps 相关的各个基础技术领域的功能、结构和过程维度的阐述简单直观而又高度实战。实验代码完整，注解清晰，实操容易上手，结果立竿见影。对数字化转型大潮中的网工们和相关技术团队的管理者们来说，本书不可不察。

——张宇峰，互联港湾 CTO





## 前言 Preface

### 为什么要写这本书

清晨，我们做的第一件事是什么？睁开眼。睁开眼看手机里的朋友圈是否有更新，昨晚下的单是否已经安排送货，今天的天气是否依旧晴朗。而这些信息的更新都是通过互联网传递到你的手机上。在很多人眼里，手机有电而没有网络是一件非常痛苦的事情。互联网在中国的发展也就是 20 来年的事，但它已经渗透到了我们工作、学习和生活的方方面面。网络是新时代的基础设施，无论上面有多么丰富多彩的应用软件，它们都离不开网络。这些年，应用软件的迭代速度非常快。而网络在这几十年中却没有发生多大的变化（虽然网络带宽一直在指数级增长）。特别是网络工程师们日常的工作似乎还是和 10 年前甚至 20 年前一样。虽然，这几年 SDN（Software Defined Networks）在快速发展，但是物理网络仍然没有发生多大的变化。大量的网络工程师还是通过 Telnet 或 SSH 登录到网络设备上，然后一条一条地敲击各种各样的命令。应用软件越来越多，应用软件生命周期越来越短。这对网络提出了很多的挑战，网络工程师的工作压力也是直线上升。这几年随着上层应用 DevOps 思想的发展，网络自动化的需求也在不断提升。那些安分守己的传统网络工程师面临着转型的痛苦。

我是一个和网络打交道 20 来年的传统网络工程师，但我一直是一个不安分守己且会偷懒的人。早在我大学期间，为了和同寝室的同学一起玩一款叫“红色警戒”的游戏而接触了网络。从两台电脑之间使用串口互联进行对战，到使用同轴电缆后 8 个同学可以在一个地图中互相厮杀，再到 1999 年通过双绞线接入互联网。那个时候，几个寝室的双绞线都汇聚到了我们寝室，我不知不觉也成了 96 级化学系的网络管理员。日常的“工作”就是帮同学看看网络怎么不通了；谁的 IP 地址又和谁冲突了；如何从其他同学的电脑里复制一些电脑游戏等。活脱脱就是一个小型网吧的工作人员。随着 1999 年学校寝室接入了互联网，出于对“工作”的热情，我开始用 Linux 自己搭建一些服务，比如 DHCP、DNS、FTP、BBS 等。慢慢又干起了系统管理员的“工作”。

在千禧年（2000 年）的毕业季，我的第一份工作是在一家大型的纺织公司做系统管理







员和 DBA。这份工作和化学没有任何的关系。而日常的工作就是帮助新员工开账号，每天备份那些数据库的数据到磁带中。为了减少自己日常的工作就开始写一些自动化的脚本。其实，当时就是为了每天能偷点懒。开一个账号，懒得去点那么多次的鼠标。每天的备份任务，懒得去一个个地核对和比较，而是让脚本自己去核对，自己去比较，然后把检查后的结果发送 E-mail 给我。

2003 年考完 CCIE 后到一家为中国电信服务的系统集成公司。在这家公司有幸参与了中国电信 CN2 (ChinaNet2) 的建设工作。在网络建设的初期有大量的设备配置需要增加和修改。纯手工的操作让我觉得痛苦不堪，此时又萌生了“偷懒”的思想。我开始用 Python、Perl 等语言写了一些脚本用于设备配置的生成和修改。当时设备并没有丰富的 API 接口，大部分都是用 Telnet 模拟登录来操作设备。

2007 年我进入了 Juniper 工作，在这里接触了更多的网络自动化的内容，也写了很多自动化脚本来操作网络设备。比如，2008 年考完 JNCIE 后，有幸做了一年多的中国区 JNCIE 考官。JNCIE 的考官除了要发卷子外，还需要负责给考生判卷。也是为了“偷点懒”写了一些自己用的脚本提高判卷的效率。2009 年开始学习 JUNOScript (一种可以运行在 JUNOS 上的脚本语言)，用 JUNOScript 来实现一些特殊的功能或者对命令进行重新格式化的输出。2010 年后由于需要经常参加设备的测试，开始使用 Python 等语言对 JUNOS 设备基于 NETCONF 协议进行操作。

2014 年到 2016 年，我先后在两家互联网公司做网络工程师，负责网络的规划与运维工作。由于互联网公司自身的产品迭代速度很快，对网络的适配性也提出了更多的需求。虽然在互联网公司有很多的程序员，但大部分的程序员对网络和网络设备的理解远逊于网络工程师。这就导致了网络自动化的开发工作比较难推进。因此，我结合自己的编程能力和对网络的理解开始用代码去实现网络自动化的任务。

从 2016 年到现在，我一直在 Cisco 工作。在这里我接触到了 DevNet (<https://developer.cisco.com>)。在 DevNet 的网站上我看到和学习了很多关于基础网络设备的编程知识。在 2016 年，Cisco 发布了思科全数字化网络架构 (思科 DNA)，这个平台不仅提供了实现全数字化的路线图，而且为网络工程师提供了网络自动化和网络安全的途径。这个平台的很多理念和架构为我写这本书提供了很多的帮助。

在这 20 来年的时间里，我积累了一些使用程序来操作网络设备的经验。一方面是想把这些经验分享给大家；另一方面也是想帮助那些想转型的传统网络工程师。这就是我写这本书的初衷。另外，我还想告诉广大的网络工程师们开发一个小工具用来管理设备其实并没有那么难。对于我这样一个非软件专业的人而言并没有觉得吃力，反而在开发中获得了更多的自信，也偷了“懒”。

最后，希望这本书能给广大的网络工程师在转型过程中带来一些帮助，也希望大家能少走弯路。





## 本书特色

首先，本书是专门针对网络工程师而写的。书中关于 Bash 和 Python 的基本语法部分使用了网络工程师更加熟悉的内容，并且提供了一些网络设备上的运行情况。

其次，本书的重点是如何编写和网络设备相关的代码。因此，在书中提供了很多关于如何处理网络设备输出的文本的例子，以及处理网络相关的数据。

最后，本书并不是一本纯粹讲解编程的书，而是一本从理论到实践的综合书籍。

## 读者对象

- 网络架构师
- 网络运维工程师
- 网络运维开发人员
- 网络与系统管理人员
- 网络规划与设计人员
- 网络专业在校学生

## 如何阅读本书

本书分为五篇，共计 14 章内容。

第一篇为概念篇，这一篇主要讲述什么是 NetDevOps，以及如何开始 NetDevOps 实践之路，包括如下 2 章内容。

第 1 章 从 SDN 开始谈起，讲解在 SDN 的大背景下，传统的网络都发生了什么变化，而这些变化给传统网络工程师带来了哪些影响。最后介绍了什么是 NetDevOps，NetDevOps 需要我们学习什么样的技能才能胜任。

第 2 章 在业务快速迭代的推动下，传统 IP 网络的自动化需求在不断增强。大量的网络工程师面临着新的挑战。这章介绍如何从零开始逐步过渡到 NetDevOps。这章将重点讲解 4 个话题：首先，在 NetDevOps 开始之前需要做什么；其次，在进行 NetDevOps 开发时，如何选择开发语言；再次，一些常见的 NetDevOps 开源工具或平台如何选择；最后，在进行 NetDevOps 时，对网络设备有哪些要求。

第二篇为基础篇，这一篇主要介绍了如何构建 NetDevOps 的工作环境以及在这些环境中的常用工具，包括如下 4 章内容。

第 3 章 介绍在 Linux 环境下，如何使用 Linux 下的工具登录网络设备，以及使用 SSH 工具建立一些 SSH 的隧道。

第 4 章 介绍在 Linux 环境下，如何使用一些工具获取网络设备的信息，以及获取网





络的可达信息，涵盖 SNMP、traceroute、ping 等工具。

**第 5 章** 使用 Linux 中三大文本处理利器（grep、awk 和 sed）来处理网络设备输出的文本内容。这些文本内容包括命令行的输出、设备的配置以及设备的日志信息等。这些工具可以帮助网络工程师快速地获取相关的数据和信息。

**第 6 章** 在 NetDevOps 的实践过程中，我们需要搭建一些基础的服务。这些服务包括 TFTP、DNS 和 DHCP 等。在微模块流行的时代，网络工程师使用 Docker 可以快速地构建起这些基础服务。

**第三篇为提高篇**，这一篇将开始介绍编程相关的内容。这一篇都是编程的一些基础知识，包括如下 3 章内容。

**第 7 章** 这一章主要介绍 Linux 环境或网络设备上的 Bash 编程基础知识。通过 Bash 基本语法并结合一些工具，我们可以和设备进行简单的交互或处理一些数据。

**第 8 章** 这一章主要介绍 Python 的编程知识。本书的大部分编程内容都是基于 Python 语言的。因此，这一章是后续章节的基础。这一章关于 Python 的基本语法是专门为网络工程师重新编写的。使用的例子将是网络工程师比较熟悉的内容。

**第 9 章** 我们在和网络设备进行交互或者进行网络相关的编程时，经常需要处理一些常用的数据类型，这些数据类型包括 JSON、XML、YAML 和 YANG。熟练掌握这些数据类型的处理是编程的基础。在这章，我们将介绍上述这四种数据类型的常用处理方法。

**第四篇为实践篇**，这一篇将通过一些实际的例子来介绍，包括如下 3 章内容。

**第 10 章** NetDevOps 必然需要和网络设备进行交互，从而获得我们需要的数据。本章将介绍三种常见的连接网络设备的方法，它们分别是：命令行登录、NETCONF 以及 REST。

**第 11 章** 连接到网络设备后就可以获取很多的信息，其中通过命令行获取的数据大部分是半结构化的数据。这些半结构化的数据需要进行结构化处理。这一章将通过几个 Python 的模块来处理这些数据。

**第 12 章** 我们在处理网络相关数据时，有两种常见且特殊的数据需要处理，它们分别是网络地址和网络拓扑数据。同样，我们将通过几个 Python 的模块来处理这些数据。

**第五部分为案例篇**，这一篇将介绍 3 个常见的案例来帮助大家更好地了解和掌握 NetDevOps 的相关内容，包括如下两章内容。

**第 13 章** 众所周知，绝大多数的网络设备都会有配置文件，获取和管理这些配置文件是 NetDevOps 工作的基础。通过程序化的方式自动地获取这些配置就打通了程序和网络设备之间的通道，这是后续获取更多信息的基础。另外，网络设备的配置文件也是最需要且被优先管理的内容，这些内容的版本管理也是非常重要的。本章将通过网络设备的配置管理案例来描述如何多厂家、并发地与网络设备进行数据交互。

**第 14 章** 网络运维与管理的独特之处是，该工作是基于网络拓扑的。获取和处理网络拓扑是基本功能。该章通过两个小的案例来介绍，它们分别是：基于 ISIS 协议来获取网



络拓扑并进行简单的网络拓扑分析；使用 BGP 协议进行简单的网络流量调度。

其中，本书的第 2 章、第 8 章、第 9 章、第 10 章、第 11 章是重点。如果你有一定的 Python 编程基础，那么可以参考第 9 章及之后的章节，这些章节提供了 Python 用于网络管理与维护常用的一些模块，这些模块可以提高你的工作效率。如果你是一位传统的网络工程师且对编程和 Linux 环境不是十分了解，请从本书的开头读起。笔者希望通过本书的内容能循序渐进地带领大家走上 NetDevOps 之路。

## 勘误和支持

由于笔者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果读者朋友有更多的宝贵意见，欢迎你发送邮件到 [netdevops@hotmail.com](mailto:netdevops@hotmail.com) 联系我。本书的大部分代码示例都放在 GitHub 上，其地址为 [https://github.com/netdevops-engineer/newbie\\_book](https://github.com/netdevops-engineer/newbie_book)。期待能够得到大家的真挚反馈，在技术之路上互勉共进。

## 特别致谢

这里要特别感谢毛厚君先生，他是这本书的第一位读者，不但给了我很多的宝贵建议，而且帮我润色了全书的文字。如果没有他的帮助我想是很难完成这本书的。

## 致谢

在本书的写作过程中得到了很多同事和朋友的支持与帮助。没有你们的支持与帮助，本书将难以如期完成。

在本书的写作过程中需要实验环境，感谢徐晓东先生为我提供了便利。

感谢思科同事们的支持和鼓励，他们是方芳女士、徐志骏先生、杨骏先生、刘佳女士等。

感谢身在美国的朋友杨文嘉先生提供了关于 Arista 产品和技术的相关信息。

最后，我要特别感谢我的家人，我为写作这本书，牺牲了很多陪伴他们的时间，但也正因为有了他们的付出与支持，我才能坚持写下去。

谨以此书献给我最亲爱的人，以及众多的网络工程师朋友们！

余 欣

## 目 录 Contents

本书赞誉

前言

### 第一篇 概念篇

#### 第 1 章 NetDevOps 理念与要义..... 2

##### 1.1 从 SDN 开始说起..... 2

###### 1.1.1 OpenFlow 打开了新的一扇窗..... 3

###### 1.1.2 简单聊聊 SDN 控制器..... 4

###### 1.1.3 NFV..... 5

###### 1.1.4 云和 SDN..... 6

###### 1.1.5 SD-WAN..... 8

##### 1.2 NetDevOps, 你需要知道的事..... 8

###### 1.2.1 什么是 NetDevOps..... 8

###### 1.2.2 NetDevOps 适用环境..... 9

###### 1.2.3 为什么我们需要 NetDevOps..... 10

###### 1.2.4 NetDevOps 需要什么样的人..... 10

##### 1.3 小结..... 11

#### 第 2 章 如何开始 NetDevOps..... 12

##### 2.1 文档内容与版本管理..... 12

###### 2.1.1 版本管理的重要性..... 13

###### 2.1.2 需要管理哪些文档..... 13

###### 2.1.3 如何实施版本管理..... 14

###### 2.1.4 版本管理的工具..... 16

##### 2.2 编程语言的选择..... 17

###### 2.2.1 程序语言的选择..... 17

###### 2.2.2 数据描述语言的选择..... 18

##### 2.3 自动化工具的选择..... 22

###### 2.3.1 Ansible..... 22

###### 2.3.2 Puppet..... 23

###### 2.3.3 Chef..... 23

###### 2.3.4 SaltStack..... 24

###### 2.3.5 如何选择..... 24

##### 2.4 网络设备的编程接口..... 25

###### 2.4.1 网络设备接口的分类..... 25

###### 2.4.2 网络设备编程接口的特征..... 27

##### 2.5 小结..... 29

### 第二篇 基础篇

#### 第 3 章 认识命令行工具..... 32

##### 3.1 用 screen 实现终端的会话管理..... 33

###### 3.1.1 安装 screen..... 34

###### 3.1.2 screen 基本语法..... 34

###### 3.1.3 screen 基本操作..... 35

3.1.4 定制你的 screen .....	36	第 5 章 处理网络设备输出的文本 .....	70
3.1.5 用 screen 连接串口 .....	36	5.1 正则表达式基础 .....	70
3.1.6 管理 screen 的日志 .....	38	5.1.1 正则表达式到底是什么 .....	71
3.1.7 多人共享一个会话 .....	38	5.1.2 单字符的匹配 .....	71
3.2 用 Telnet 和 SSH 管理设备 .....	39	5.1.3 多字符的匹配与次数匹配 .....	75
3.2.1 Telnet .....	39	5.1.4 在网络设备上的正则表达式 .....	77
3.2.2 SSH 介绍 .....	40	5.2 使用 grep 进行搜索与获取信息 .....	78
3.2.3 SSH 的基本使用 .....	40	5.2.1 什么是 grep .....	78
3.2.4 利用 SSH 远程执行命令 .....	42	5.2.2 命令选项的解释 .....	78
3.2.5 SSH 客户端常用配置 .....	44	5.2.3 匹配控制 .....	80
3.2.6 使用密钥登录设备 .....	45	5.2.4 输出结果控制 .....	81
3.2.7 使用 scp 进行文件传输 .....	47	5.2.5 输入控制 .....	83
3.2.8 利用 SSH 端口隧道转发功能 .....	48	5.3 使用 awk 进行文本处理 .....	84
3.2.9 利用 SSH 做 Socket 代理 .....	50	5.3.1 认识一下 awk .....	84
3.3 小结 .....	50	5.3.2 awk 的执行方式与语法 .....	84
第 4 章 Linux 下的一些常用工具 .....	52	5.3.3 截取部分信息 .....	85
4.1 SNMP .....	53	5.3.4 使用内置变量 .....	86
4.1.1 SNMP 简介 .....	53	5.3.5 对特定内容进行统计分析 .....	86
4.1.2 常见设备的 SNMP 配置 .....	54	5.3.6 多文件操作 .....	88
4.1.3 SNMP 工具 .....	56	5.4 使用 sed 进行文本编辑 .....	89
4.2 网络可达性检测工具 .....	58	5.4.1 什么是 sed .....	89
4.2.1 Nmap .....	59	5.4.2 sed 语法简介 .....	89
4.2.2 Nping .....	62	5.4.3 删除文件中的指定信息 .....	90
4.2.3 iPerf .....	63	5.4.4 在文件中进行查找替换 .....	91
4.2.4 Fping .....	64	5.4.5 在文件中插入内容 .....	92
4.3 MTR .....	65	5.5 文本编辑工具 vi 和 vim .....	92
4.4 其他工具 .....	66	5.5.1 vi 和 vim 简介 .....	92
4.4.1 watch .....	66	5.5.2 vim 编辑器的模式 .....	93
4.4.2 Wget .....	68	5.6 小结 .....	97
4.4.3 CURL .....	68	第 6 章 常用基础服务搭建 .....	99
4.5 小结 .....	69	6.1 Docker 基础 .....	100



6.1.1	什么是 Docker	100
6.1.2	Docker 的基本概念	101
6.1.3	Docker 的运行环境	104
6.1.4	启动 Docker 镜像	105
6.1.5	构建 Docker 镜像	106
6.2	TFTP 服务器	110
6.2.1	定制一个 TFTP 服务镜像	111
6.2.2	启动一个 TFTP 服务器的容器	112
6.2.3	服务的检查	112
6.3	DNS 服务器	113
6.3.1	构建 DNS 镜像	113
6.3.2	启动和配置 DNS	114
6.3.3	用 DNS 记录设备的接口与 IP 的对应关系	115
6.4	搭建 DHCP 服务器	118
6.4.1	构建 DHCP 镜像	119
6.4.2	启动和配置 DHCP 服务	120
6.5	小结	121

### 第三篇 提高篇

第 7 章	Linux 编程基础	124
7.1	Bash 编程基础	124
7.2	第一个 Bash 程序	125
7.3	变量	126
7.4	数组	128
7.4.1	定义数组	128
7.4.2	数组取值	129
7.4.3	获取数组的长度	129
7.4.4	截取数组的内容	130
7.4.5	替换元素中的内容	130
7.4.6	删除数组中的元素或者数组	130

7.5	运算符	131
7.5.1	算术运算符	131
7.5.2	位运算符	132
7.5.3	自增 / 自减运算	136
7.6	测试	136
7.6.1	测试语法的结构	136
7.6.2	文件测试	136
7.6.3	整数测试	138
7.6.4	字符串测试	138
7.6.5	逻辑关系	139
7.7	判断结构	140
7.7.1	if 结构	140
7.7.2	case 结构	141
7.8	循环结构	141
7.8.1	for 结构	141
7.8.2	while 结构	143
7.8.3	until 结构	144
7.8.4	select 结构	144
7.9	函数	145
7.10	用 expect 实现与设备的交互式 操作	147
7.10.1	expect 简介	147
7.10.2	用 expect 实现与设备的交互	148
7.10.3	用 expect 实现批量备份设备 配置	150
7.11	网络设备上的 Bash	152
7.12	小结	154

第 8 章	Python 编程基础	155
8.1	Python 简介	155
8.1.1	Python 的版本差异	155
8.1.2	主机与网络设备上的 Python	156

8.1.3	构建 Python 运行环境	158
8.1.4	缩进在 Python 中的重要性	161
8.2	基本数据类型	161
8.2.1	数字	162
8.2.2	列表	163
8.2.3	元组	166
8.2.4	字符串	167
8.2.5	字典	170
8.2.6	集合	173
8.3	基本结构	175
8.3.1	选择结构	175
8.3.2	循环结构	177
8.4	函数	181
8.4.1	函数的定义	181
8.4.2	函数的参数	183
8.5	对象	186
8.5.1	什么是对象	186
8.5.2	对象的属性和方法	186
8.5.3	创建对象	187
8.5.4	对象的继承	188
8.6	模块	190
8.6.1	什么是模块	190
8.6.2	如何使用模块	190
8.7	小结	191

## 第 9 章 常用数据类型与数据结构

### 定义

9.1	JSON	192
9.1.1	JSON 简介	193
9.1.2	网络设备上的 JSON	194
9.1.3	JSON-RPC	196
9.1.4	用 Python 处理 JSON	196

9.2	XML	198
9.2.1	XML 简介	198
9.2.2	XML Schema	200
9.2.3	NETCONF	201
9.2.4	用 Python 处理 XML	202
9.3	YAML	204
9.3.1	YAML 简介	205
9.3.2	YAML 语法	206
9.3.3	用 Python 处理 YAML	207
9.4	YANG	208
9.4.1	YANG 简介	208
9.4.2	YANG 语法	211
9.4.3	OpenConfig	214
9.4.4	Pyang 工具	214
9.5	小结	216

## 第四篇 实践篇

## 第 10 章 网络设备的连接与登录

10.1	命令行方式登录	218
10.1.1	telnetlib	219
10.1.2	paramiko	221
10.1.3	netmiko	224
10.1.4	pexpect	227
10.2	通过 NETCONF 连接到网络 设备	231
10.2.1	安装 ncclient	231
10.2.2	获取配置信息	231
10.2.3	获取接口信息	233
10.3	REST	235
10.3.1	测试 REST 接口	236
10.3.2	安装 requests 模块	237

10.3.3 使用 HTTP get 方法	237	12.2.5 地址的聚合	267
10.3.4 使用 HTTP post 方法	238	12.2.6 IPv6 地址	268
10.4 小结	239	12.2.7 使用 netaddr 处理 MAC 地址	268
<b>第 11 章 命令行文本处理</b>	240	12.3 使用 ipaddr 处理网络地址	269
11.1 命令行文本输出	240	12.4 网络拓扑的处理	271
11.1.1 关于 TextFSM	241	12.4.1 描述一个网络拓扑	271
11.1.2 安装 TextFSM	241	12.4.2 最短路径的计算	273
11.1.3 TextFSM 模板	242	12.4.3 可用路径的计算	276
11.1.4 如何编写 TextFSM 模板	243	12.5 小结	278
11.1.5 在 Python 代码中使用 TextFSM	248		
11.2 Cisco 配置类型	249	<b>第五篇 案例篇</b>	
11.2.1 ciscoconfparse 模块	249	<b>第 13 章 网络设备的配置管理</b>	280
11.2.2 安装模块	250	13.1 环境的准备	280
11.2.3 获取配置内容	251	13.1.1 测试拓扑说明	280
11.2.4 修改设备配置	252	13.1.2 Linux 服务器的准备	281
11.2.5 配置审计	253	13.2 网络设备的配置获取	282
11.3 JUNOS 配置类型	254	13.2.1 登录网络设备	282
11.3.1 层次化配置	255	13.2.2 处理多厂家问题	287
11.3.2 set 命令行配置	256	13.2.3 处理并行问题	290
11.4 小结	259	13.3 网络设备的配置版本管理	295
<b>第 12 章 网络特有数据类型处理</b>	260	13.3.1 用 git 创建一个本地设备配置 管理仓库	296
12.1 Jupyter	260	13.3.2 保存设备配置文件到本地 仓库	296
12.1.1 安装 Jupyter	260	13.3.3 使用 git 检查版本信息	297
12.1.2 启动 Jupyter	261	13.4 小结	299
12.1.3 使用 Jupyter	263		
12.2 使用 netaddr 处理网络地址	264	<b>第 14 章 网络拓扑的处理与应用</b>	300
12.2.1 安装 netaddr 模块	264	14.1 环境的准备	300
12.2.2 IP 地址的基本属性	264		
12.2.3 处理 IP 地址的基本方法	265		
12.2.4 IP 地址的加减法	266		



14.1.1	测试拓扑说明 .....	300	14.3.1	基本信息 .....	315
14.1.2	Linux 服务器的准备 .....	300	14.3.2	路径计算 .....	316
14.2	网络拓扑的获取与分析 .....	304	14.3.3	BGP 服务 .....	318
14.2.1	物理拓扑的获取 .....	304	14.3.4	调用 BGP HTTP API .....	324
14.2.2	ISIS 协议拓扑的获取 .....	311	14.3.5	结果测试 .....	324
14.2.3	网络拓扑的路径分析 .....	313	14.4	小结 .....	325
14.3	网络流量工程应用 .....	314			



## 第一篇 *Part 1*

# 概念篇

欢迎来到《NetDevOps 入门与实践》。

在第 1 章，我们将讨论一下 NetDevOps 的一些基本概念以及 NetDevOps 与 SDN（软件定义网络）相关的一些内容，并且介绍一下为什么需要 NetDevOps。

第 2 章主要涉及如何开始 NetDevOps。随着上层业务的推动，传统的 IP 网络面临着前所未有的挑战，网络的自动化需求在不断增强，如何从零开始逐步向 NetDevOps 过渡，这是目前大量传统网络工程师的困惑所在。本章将涉及以下几个话题：

- ☐ 开始 NetDevOps 之前首先需要做什么；
  - ☐ 关于 NetDevOps 语言的选择问题；
  - ☐ 一些常见的 NetDevOps 开源工具的选择；
  - ☐ 网络设备需要具备什么能力以及如何来管理它们。
- .....

# NetDevOps 理念与要义

自从 20 世纪八九十年代 Internet（互联网）出现以来，IP 网络在全球出现了爆发式发展。近几年来，随着大量计算机网络应用，特别是云计算、大数据以及互联网+在各行各业的渗透，网络变得无处不在。虽然网络不是万能的，但是现代社会没有网络几乎已经变成万万不能了。各种各样纷繁复杂的应用都承载在 IP 网络之上，网络规模变得越来越大，网络结构也变得越来越复杂。这些变化给网络工程师带来的压力在逐年增加。如何提高网络运维的效率、提升网络操作准确性以及网络业务可用性是网络工程师一直在苦苦探索的方向。

对于 NetDevOps，你需要向广大网络工程师、网络规划人员以及大部分运维平台开发工程师解释清楚它到底是什么、它包含哪些内容，以及它和这几年火热的 SDN（软件定义网络）有什么关系。

## 1.1 从 SDN 开始说起

在这几年的 IT 圈子中，最火的名词有云计算、大数据以及人工智能（AI）。在网络，特别是基础网络圈子里，最火的也许就是 SDN 与 NFV 了。我们在全球 IT 相关的大会中都可以看到它们的身影。在国内，网络厂商的技术交流会，以及互联网公司、运营商、企业的技术分享会都会提到 SDN 和 NFV 相关的解决方案。在网络运营与维护工程师聚集的论坛里也经常看到它们的主题。比如，NANOG（北美网络运维论坛 <https://www.nanog.org>）也有 SDN 与 NFV 相关的内容。作为在网络行业中工作了近 20 年的网络工程师，笔者和很多在这个行业中的网络工程师们一样，对于 SDN，也许一开始是拒绝的。但是随着 SDN 的声音越来越大，参与的人也越来越多，很多像笔者一样的网络工程师也开始关注和参与到这个领域。



那么 SDN 究竟和 NetDevOps 有什么关系或关联呢？正是 SDN 的出现和发展，推动了传统网络设备管理维护方式的变革：一方面，SDN 让网络工程师不再只依赖 CLI 命令行（部分 Web）方式对设备进行逐台的操作管理，毕竟逐台通过 CLI 的管理方式是一种非常低效且容易出错的运维方式；另一方面，SDN 推动了传统网络设备厂商自身的变革，各厂商开放出更多的 API 接口，供用户侧进行自动化的编排和调用。自动化这几年的呼声越来越大，自动化上线、自动化部署、自动化运维、自动化修复，甚至还可以和业务进一步结合。实际上，NetDevOps 追求的目标就是网络资源的接口化、自动化以及智能化。借助于 NetDevOps，我们可以提高运维的工作效率，提升业务部署的准确性，提供网络资源的可编程性。

在正式进入 NetDevOps 之前，我们先来了解一下 SDN 的发展状况。无论是 SDN 具体的技术细节，还是 SDN 的建设思想，SDN 对全球的学术研究机构、标准化组织以及设备厂商都已经产生了深远的影响。了解 SDN 技术的概貌及其相关的实现架构，有助于我们理解 NetDevOps。

### 1.1.1 OpenFlow 打开了新的一扇窗

对于 OpenFlow，我想大家应该并不陌生。有人认为 OpenFlow 是 SDN 的“Hello World”，也有人觉得它给网络带来了新的活力。众所周知，Martin Casado 在斯坦福大学读博士期间领导并开发了 OpenFlow 协议，当时他的博士导师是 Nick McKeown。这个协议不仅定义了网络设备数据转发平面的内容，同时也定义了控制平面的内容。简单来说，通过 OpenFlow，可以把控制平面从硬件中剥离出来，可以开发出更加智能、更加集中的控制中心。而转发平面把原来网络设备 ASIC 等芯片进行了更加高级的抽象化处理，让开发人员不用深入了解硬件底层的处理方式，就能够获得硬件带来的性能。《OF-CONFIG 1.2 ONF TS-0161》<sup>①</sup>白皮书给出 OpenFlow 高度抽象的逻辑图（见图 1-1）。从这个图中，我们可以清晰地看出：OpenFlow 协议是 OpenFlow 控制器与 OpenFlow 交换机之间通信的协议，而交换机的其他操作使用了 OF-Config 这个接口协议。如果我们把这个结构映射到传统的路由器或者交换机中，OpenFlow 协议可以类比为传统设备的 RIB（Routing Information Base，路由信息表）与 FIB（Forwarding Information Base，转发信息表）之间的协议。

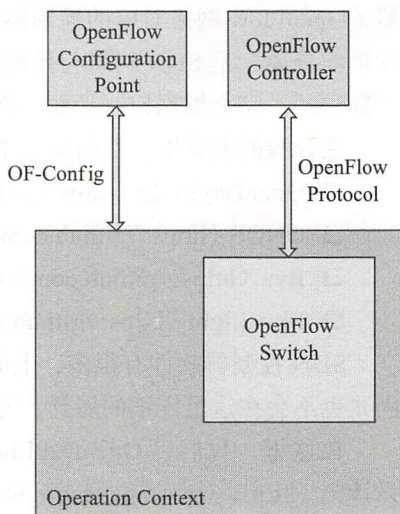


图 1-1 OpenFlow 逻辑图

① <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf>。



对于传统的网络设备，这里的通信协议往往是私有的，并没有开放给普通的使用者；而 OpenFlow 协议则定义了这部分内容。除了 OpenFlow 协议，OF-Config 定义的是如何管理和运维网络设备。OF-Config 定义了机器与机器之间的交互方式。它和设备之间的通信使用 NETCONF 作为底层的通信协议，使用了大量的 YANG 模型对数据结构进行了定义，并且给出了 UML (Unified Modeling Language, 统一建模语言) 的结构。

OpenFlow 协议本身并不是一场变革，它所描述的方式、方法不是传统网络工程师所熟悉的领域，反而是软件工程师所熟悉的领域，它从另外一个视角来描述网络的管理与维护工作。它让传统的人机交互命令演变为控制器与机器之间的交互，从而可以实现集中化、统一化和程序化的网络管理维护方式。

NetDevOps 的初衷也是希望通过机器与机器之间的交互来实现更加高效、更加准确的网络管理，而不只限于对网络设备的管理；另外，也希望通过对网络资源进行二次封装，从而为上层应用提供简单灵活的编程接口。

### 1.1.2 简单聊聊 SDN 控制器

接着前面谈到的 OpenFlow 相关的一些内容。OpenFlow 自身并没有定义和解决控制平面的问题。在 OpenFlow 应用场景中，我们通过 SDN 控制器来保证运行 OpenFlow 的网络能正常地运作。SDN 控制器完成的是控制平面的工作，它是通过软件来实现的。为了满足不同场景的应用需求，控制器的内容会更加抽象化，在表述控制器功能的时候也更加软件工程化。在 SDN 网络（包括 OpenFlow 网络）中，和智能相关的内容大部分由 SDN 控制器（OpenFlow 网络对应的控制器通常称为 OpenFlow 控制器）完成和实现。“控制平面与转发平面分离”是 SDN 网络与传统网络的主要区别。转发平面常常被定义为毫无智能可言的“傻终端”，完全根据控制平面的指令来进行转发。

在开源的世界里，目前比较主流的 SDN 控制器如下：

- ❑ OpenDayLight (<https://github.com/.opendaylight>);
- ❑ ONOS (<https://github.com/opennetworkinglab/onos>);
- ❑ Ryu (<https://github.com/osrg/ryu>);
- ❑ Floodlight (<https://github.com/floodlight/floodlight>)。

SDN 控制器软件有很多，上面列出的只是其中很少的一部分。SDN 并不是本书的重点，因此也不会在这里详细的展开，有兴趣的读者可以找相关的书籍或者文章进行了解。

在这里，我们以 OpenDayLight (简称 ODL) 为例子简单地看一下 SDN 控制器的框架设计图 (<https://www.opendaylight.org/odlboron> 网站提供了更加清晰的图片)。

从 OpenDayLight 版本框架结构图 (见图 1-2) 中，可以清晰地看出以下几点。

首先，最下方是网络设备，既包括具备 OpenFlow 功能的设备，也包括 Open vSwitch 这样纯软件的设备，还包括了一些其他的硬件物理设备。

其次，ODL 的南向接口（即控制器与网元设备通信的接口）不单单只有 OpenFlow 与





OF-Config, 还包括 NETCONF、BGP、PCPE、SNMP 等多种接口。ODL 实现了基于南向接口的网络服务抽象层和基于抽象服务的接口转换。通过这一层的转换, 能够让网络设备提供的描述方式和能力转化为面向业务的描述方式及接口。这就方便了应用层的开发人员进行网络相关应用程序的开发, 他们并不需要深入了解网络底层的内容。

最后, 控制器提供了北向 API 接口 (有 RESTful、RESTCONF、NETCONF 等接口类型), 这是应用开发人员编程所使用的接口。其他更加上层的系统或者平台可以通过调用这些接口来完成其开发, 完成应用对网络的调度和管理。ODL 提供了 GUI 的开发框架, 应用开发人员也可以基于这个开发框架来开发应用程序, 并实现对网络的管理和调用。

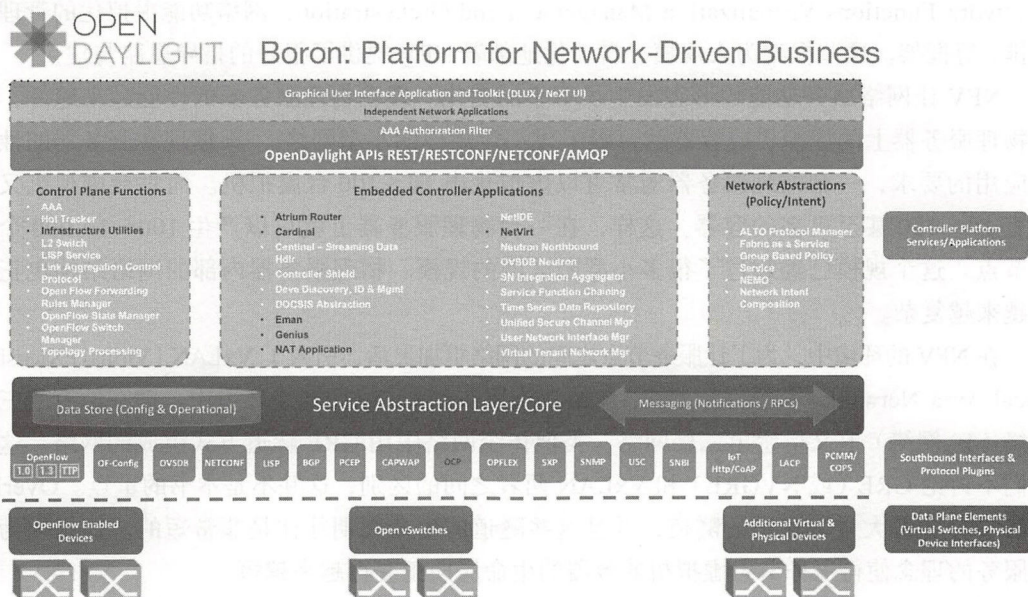


图 1-2 OpenDayLight 硼版本框架结构图

总之, SDN 控制器软件是应用开发人员和网络设备 (无论是硬件的还是软件的) 之间的桥梁。通过 SDN 控制器, 应用开发人员可以更多地专注于业务逻辑, 从而开发与网络相关的应用程序。因此, 在 SDN 控制器设计的时候, 无论是南向接口还是北向接口都会更加关注机器与机器之间的接口。这也许是很多传统网络工程师感觉 SDN 控制器比较难学习和难以掌握的地方。正像前面提到的, 网络工程师擅长的还是人与机器交互的界面, 这也导致了网络工程师在接触 SDN 控制器时, 通常也只是关注 SDN 控制器所提供的 Web UI 功能的原因。这样的思路需要转变, 无论是 SDN 还是 NetDevOps, 都应该更加关注机器和机器之间的接口部分。

### 1.1.3 NFV

随着 SDN (软件定义网络) 的发展, NFV (Network Functions Virtualization, 网络功能





虚拟化)在SDN领域也越来越多地被提及。NFV是ETSI(European Telecommunications Standards Institute, 欧洲电信标准协会)在2012年10月发布的白皮书<sup>①</sup>中定义的。但是, NFV这样的产品形态并不是2012年后才有。例如, 2006年春季推出的Vyatta OFR<sup>②</sup>是一个软件的路由器, 它可以运行在很多的虚拟化的平台上。又如, 2009年, Cisco推出Nexus 1000v<sup>③</sup>的虚拟交换机产品, 它可以运行在VMware等虚拟化平台上, 提供软件交换机的功能。另外, 防火墙、负载均衡等领域都有虚拟化软件的产品, 如Juniper的vSRX<sup>④</sup>虚拟化和cSRX<sup>⑤</sup>容器化的软件防火墙产品。ETSI除了概念化了NFV, 还定义了NFVI(Network Functions Virtualization Infrastructure, 网络功能虚拟化的基础设施)以及NFV-MANO(Network Functions Virtualization Management and Orchestration, 网络功能虚拟化的管理与编排)等框架。这些框架为大规模、自动化地部署NFV提供了很好的指导思路与建议。

NFV让网络服务功能的形态发生了不小的变化。现在物理服务器的性能越来越好, 单台物理服务器上的虚拟机或容器就可以组成一定规模的小型网络。根据现在服务器的性能及应用的要求, 一台物理服务器通常可以虚拟化出10~100台虚拟机。而每台虚拟机又可以有10~100甚至更多的容器。这样, 在一台物理服务器上就可以产生100~10 000个服务节点。这个规模已经超过了很多小型园区网的规模。物理服务器内部的网络结构也正变得越来越复杂。

在NFV的环境中, 为了让服务节点之间的网络更加灵活, 引入了VxLAN(Virtual eXtensible Local Area Network)等技术。VxLAN通常应用在Overlay的网络环境中, 即在原有的三层网络(IP网络)上又衍生出二层网络, 类似传统网络中用GRE隧道方式组成的网络。这里我们不讨论GRE(或NVGRE)和VxLAN两者之间的区别, 这并不是本书的重点。Overlay网络中存在着大量非状态的隧道, 并且这些隧道的生命周期往往是非常短的。这是因为按需服务的理念使得服务节点虚拟机或容器的生命周期正变得越来越短。

网络功能的软件化与虚拟化使得网络功能部署更加灵活和快捷, 这势必会促使NFV软件大规模发展和应用, 软件网元数量将会大大超过现在物理网元设备。同时, 业务需求的频繁调整 and 变化, 将使得Overlay网络拓扑的生命周期越来越短。管理和维护由软件网元组成的网络将是一个很大的挑战。

### 1.1.4 云和SDN

“云”应该是大家再熟悉不过的概念了。现在有很多的公有云平台, 如美国亚马逊的

---

① [https://portal.etsi.org/NFV/NFV\\_White\\_Paper.pdf](https://portal.etsi.org/NFV/NFV_White_Paper.pdf)。

② <https://wiki.vyos.net/wiki/Vyatta>。

③ <https://communities.cisco.com/community/technology/datacenter/data-center-networking/nexus1000v/blog/2009/7>。

④ <https://www.juniper.net/us/en/products-services/security/srx-series/vsrx>。

⑤ <https://www.juniper.net/us/en/products-services/security/srx-series/csrx>。



AWS 云平台、微软的 Azure 云平台以及谷歌的云平台等；国内有阿里巴巴的阿里云平台、腾讯的云平台、百度的云平台以及青云等。另外，大量企业使用了 OpenStack、VMware 等搭建企业的私有云或公有云平台。在私有云和公有云之间又存在很多混合云的环境。从网络的视角来看，这些云的通信可以分为两大类：一类是云内部的通信。这通常是在一个 DC（Data Center，数据中心）内部的通信。另一类是云与云之间的通信。这里有些是私有云之间的通信，有些是公有云平台之间的通信，有些是公有云与私有云之间的通信。这些云之间的通信有些是专门的线路互联，例如，亚马逊的 AWS 平台就提供直接网络互联的服务。有些是没有专门的链路进行连接的，它们之间通过现在的互联网进行互联。

当计算、存储等资源云化后，这些资源的弹性变大了。换个角度来看，这些资源池所提供的服务节点虚拟机的生命周期也变短了，需要时启用，不需要时释放。这些资源都是网络中的节点，云越多，可用的资源也会越多，并且所有这些资源都会愈加依赖网络进行通信。散布在各地的云资源需要通过网络进行动态的组合，这势必会迫使网络具备动态的调整能力。连接云节点与云节点之间的网络无论是通过底层的物理网络来实现，还是通过上层的软件网络来实现，都需要具有一个供机器管理调用的接口，才有可能实现对网络的监控、创建、修改以及撤销等操作功能。

在网络中传送的数据最终还是会落到物理连接上。因此，物理网络（有时候也把它称为传统网络）同样需要实现更多机器与机器的通信接口。正是像云业务这样上层应用的不断“推动”，才使得网络的接口和自动化能力需要不断提升，从而达到更高的业务敏捷性。同时，软件定义网络技术的发展“拉动”了网络自身的变革。我们可以清晰地看到，网络的变革体现在以下几个方面。

首先，网络管理接口数据结构化，这里比较有代表性的是 NETCONF 协议的定义以及 YANG 模型对网络设备的配置与信息输出结果的结构化定义。关于 NETCONF 协议与 YANG 模型这两部分的内容，在后续的章节中会比较详细的介绍。

其次，在 SDN（软件定义网络）中强调转发与控制分离的架构。这种架构实际上是让硬件与软件进行了分离。软件部分的控制器在设计上提供了非常丰富的北向接口。而这些北向接口的定义，都是为了实现机器与机器（程序与程序）之间的通信。当网络设备的 API 越来越丰富时，在这样的环境中进行自动化运维就不是难事了。

大家也许注意到，刚才一直在说机器与机器通信的问题。正是类似像“云”这样的业务和 SDN 这样的技术促进了网络设备接口的升级。无论是商业公司的产品还是开源的项目，均提供了比以前更多的机器与机器交互的接口，而不再只是关注人与机器交互的接口。较典型的人与机器交互的接口是 CLI（Command Line Interface，命令行）以及 Web 界面。而这样的接口是很难与另外一台机器进行交互的。只有让更多的机器参与到业务流程中，才能提供更高的效率。网络提供的服务在业务平台的下层，如果网络能够给上层业务应用提供更多的可供机器使用的接口，网络被使用的效率自然也会得到快速提高。





### 1.1.5 SD-WAN

SDN 一直被认为缺少理想的应用场景。而自 2014 年开始出现的 SD-WAN (Software Defined-WAN) 正逐渐成为 SDN 最热门的一种应用场景。截至 2017 年, 业界对 SD-WAN 的定义还存在着一些分歧, 具体可以参考 Wikipedia (<https://en.wikipedia.org/wiki/SD-WAN>) 的内容。

在笔者看来, 网络从功能、物理规模和特点方面来区分大致可以分为以下三大类。

1) 广域网 (Wide Area Network, WAN)。Wikipedia 给出的定义如下: 广域网是指连接不同地区局域网或城域网计算机通信的远程网。广域网通常跨接很大的物理范围, 所覆盖的范围从几十千米到几千千米。由于长途链路的成本是这张网络的主要成本, 因此这张网络的拓扑结构差异性会比较大。广域网存在链路按需使用的需求, 例如早期的 ISDN 链路就是按时长进行计费的; 也存在网络流量调度的需求, 即合理地利用现有的链路。Google B4 在这个领域做了很多的尝试<sup>①</sup>。

2) 园区网络 (也称为局域网)。相对于广域网而言, 它的覆盖范围通常会比较小, 常常仅限于一幢或者几幢楼的内部, 并且其带宽相对广域网往往会大很多。园区网的主要功能是提供大量的信息点并管理这些信息点接入的用户, 这些信息点可以是一些传感器或者是人机交互的终端。其主要特点是要应对各种各样的接入方式以及安全接入的需求。园区网的网络拓扑相对比较固定, 通常是双星形的拓扑结构。

3) 数据中心网络 (Data Center Network, DCN)。这个网络的物理范围一般会更小, 只覆盖一个或数个较近的机房。这个网络的拓扑非常标准化, 目前最为流行的设计是 SPINE-LEAF 的结构<sup>②</sup>, 设计完成后几乎不会有大的变动。其主要的特点是为服务器和服务器之间的通信提供高速的带宽, 通常也称之为横向流量带宽。

而 SD-WAN 首先要解决广域网的互联互通, 其次需要满足网络的快速开通与灵活部署, 另外还要解决一些广域网优化的问题。SD-WAN 会大量引入 NFV, NFV 的灵活性在 1.1.3 节中有介绍。如何快速部署业务、如何管理网络节点、如何检测广域网的通信质量、如何优化广域网的转发路径、如何运营广域网, 这些都是 SD-WAN 需要解决的问题。作为软件定义的广域网, SD-WAN 必然会使用很多编排系统, 这些编排系统也必然会大量用到网络设备的 API。对于这样的网络, 采用 DevOps 的运维方式和业务部署将是一个不错的选择。

## 1.2 NetDevOps, 你需要知道的事

### 1.2.1 什么是 NetDevOps

NetDevOps 是网络 (Networks)、开发 (Developments) 与运维 (Operations) 三个单词

① <https://conferences.sigcomm.org/sigcomm/2013/papers/sigcomm/p3.pdf>。

② 可以参考 <http://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/white-paper-c11-737022.html>。





的复合词。很显然，这个单词已经非常清晰地表达了其所包含的内容：网络的运维工作需要开发者来一起参与进行，通过程序化的代码来完成大量运维的工作。推动这种变化（或称为演进）有来自多方面的因素：网络规模变大带来维护工作量的增加、应用快速上线、快速响应需求、自动化运维思想的驱动等，总之就是需要提高运维效率，给运维人员降低工作负荷，提升网络操作的准确性，降低误操作。从定义来看，NetDevOps 包含很多方面的内容，它既包含了技术的一些细节，也涵盖了很多非技术的内容。实际上，NetDevOps 代表了一种思路、一种方法论、一种与时俱进符合当前业务发展需要的新型的网络管理和运维手段。

本书主要从技术入手，侧重介绍实践过程中能直接应用的工具与方法，在最后会给出几个常见的应用场景的实际案例。

NetDevOps 不一定都要通过编程来实现。毕竟对于绝大多数网络工程师而言，编程属于陌生或跨界的领域，要求传统网络工程师一下子就能编写出大量代码来实现自动化的网管和运维也是比较困难的。学习和实践 NetDevOps 并不一定需要我们从头开始写每一行代码，我们可以学习和掌握一些已有的工具和现成的程序库来实现我们的想法、达到我们的目的。尤其是对于初学者而言，有现成 NetDevOps 工具能实现的就尽量不要使用编程，有现成的语言平台库能实现的功能就尽量不要去一行一行地去撰写代码。可以逐步编写更多的代码或基于已有的代码模板进行修改，这是一个循序渐进的过程，只有你一路坚持下来才能体会出这其中美妙的、得心应手的滋味。

### 1.2.2 NetDevOps 适用环境

现在得益于 IT 技术的发展，很多行业自身的自动化程度越来越高。随着人工智能的逐步普及，以后机器将会能做更多的事情。但是目前很多的网络规划与运维工程师们，特别是在国内，大家都还不太善于利用机器来帮助自己做更多的事情。在云计算的大趋势下，无论是网络的自动开通，还是网络的变更运维，都需要非常高的自动化程度。目前不论是传统的网络还是 SDN 相关的网络，都适合采用 NetDevOps 的方式进行运维管理。

在传统的网络中，设想某网络工程师管理维护了一套企业园区网，假设这个园区网从接入到汇聚再到核心和网络出口一共拥有 100 台网络设备。在日常的管理维护方面，网络工程师需要监控这些设备的运行状态，并定期做设备配置的备份或设备版本的升级；当然也会经常有新业务上线，对应网络及安全的策略变更等工作。可想而知，如果没有类似 NetDevOps 工具的帮助，此网络工程师的工作量将会非常庞大。而如果能够借助 NetDevOps 来完成此类日常事务，那么此网络工程师的工作量可以缩减为原来的十分之一，甚至更少。由机器完成此类重复类（循环类）的信息统计，其效能将万倍于人的手工操作。

在云服务商自身的 SDN 网络中，NetDevOps 已经得到较为广泛的使用，包括公司层面 DevOps 的开发（此类 NetDevOps 通常会跟公司主营业务应用以及服务器虚拟化平台进行融



合，打通业务自动化链条上的各个环节)以及网络运维部门自身的维护 NetDevOps 工具的开发。国外 Google 公司内部日常运维的网络工程师已经要求必须具备 NetDevOps 的编程能力。

NetDevOps 实际上适合运用在任何存在网络的环境里，包括传统园区网、新型数据中心，以及当下日益火热的 SD-WAN 市场。

### 1.2.3 为什么我们需要 NetDevOps

为什么我们需要 NetDevOps？原因有以下几方面。

首先，上层业务的自动化程度越来越高。这些自动化的业务推动了网络管理需适应其业务自身的快速迭代与发展。

其次，网络设备及网络技术的演进。最为典型的就是 SDN 的发展让网络设备的 API 越来越丰富，这点拉动了网络管理方式的变革——可以更多地使机器和代码参与到日常的网络管理工作之中来。

再次，网络节点数量在急剧增加，导致重复性工作越来越多。只有使用机器和代码才能快速高效地完成此类工作。最后，人工管理网络势必会不可避免地带来大量人为疏忽性错误。网络维护操作中的小失误常常会引起大范围的网络故障，因此网络维护的准确率是非常重要的。就这点而言，使用机器和代码完成自动化将会是更好的选择。

纵观当前的 IT 发展状况：快速服务是目的，自动化是手段，NetDevOps 是方法。NetDevOps 是之前游戏规则的改变者，它打破了传统 IT 各领域的边界，其本质是一种思路、一套方法论。如今许多网络和 IT 运维人员已经开始寻求和部署自动化的解决方案，并借此来提升用户体验，提升服务质量和效率。当然这些技能也会提升网络运维人员自身的竞争力。

上士闻道，勤而行之。既然已经有了这么强大、这么好用的技术和理念，我们有什么理由仍然拒之门外呢？

### 1.2.4 NetDevOps 需要什么样的人

传统的网络工程师需要掌握很多的网络基础知识。例如，OSPF、ISIS、BGP 等路由协议，还有相关设备厂家的命令行使用方式等。既然希望借助 NetDevOps 让机器和代码参与到网络工程师日常的管理与运维工作中，那么必然也需要要掌握一些 NetDevOps 相关的基础知识，如文本处理工具、网络设备的 API、一些简单的编程语言以及一些现成的自动化工具。本书的目的就是帮助传统网络工程师了解和学习这些方面的基础知识。即便是传统网络工程师不愿转向软件开发领域，但了解和掌握一些 NetDevOps 的基础知识也是很有益处的。当你和软件开发、软件设计人员进行交流时，当你希望公司开发部人员能帮你们部门开发一些系统和工具时，这些知识将会很有帮助。



## 1.3 小结

本章一开始谈了很多关于 SDN 和云相关的内容，看似和 NetDevOps 没有太多关系，但正是 SDN 和云的发展让网络 DevOps 变成了可行的且相当迫切的需求。如果不是 SDN 和云这两个技术的双重驱动力，NetDevOps 发展也许还会再晚一些。

NetDevOps 带着 DevOps 的理念与文化，在传统的网络运维领域也一样会掀起出新的浪潮。在第 2 章，我们会和大家聊聊关于如何开始 NetDevOps 的学习。





## 如何开始 NetDevOps

基于第 1 章的内容，我们可以看到 NetDevOps 是网络运维的发展趋势。那么，我们如何开始 NetDevOps 呢？我们需要做哪些准备工作呢？本章会从如下几个方面来帮助大家做好准备工作。

- ❑ 文档的管理。无论是采用传统的方式运维网络还是采用写程序的方式来维护网络，都会产生大量的文件，有效地管理这些文档和代码以及相应的版本是 NetDevOps 的开始。
- ❑ 编程语言的选择。NetDevOps 包含了一定的程序开发工作，如何选择一个适合自己或自己所在公司的编程语言是一件比较重要的事情。目前编程语言多达 5000 多种，我们怎样才能选择更适合当下的语言呢？
- ❑ 自动化平台的选择。对大多数中小企业而言，开发一套相对完整的自动化工具是一项工作量很大的工程。特别是在初期，使用现成的工具将会是一个不错的选择。但这些自动化的平台该如何选择呢？
- ❑ 我们需要什么样的编程接口。既然要面向网络设备进行编程开发，那么网络设备需要具备什么样的编程接口才会更容易实现 NetDevOps 呢？

我们在开始编程之前先关注一下上述的几个问题，这对后续的学习会有帮助。

### 2.1 文档内容与版本管理

首先需要声明的是，版本管理并不限于网络工程师们常说的“设备软件版本的管理”。这里提到的版本管理包括网络规划、网络运维涉及的所有文档、代码和设备相关的软件。具体的内容会在后续进行详细的说明。本节从以下几点进行描述：



- ❑ 版本管理的重要性；
- ❑ 网络管理中哪些内容需要版本管理；
- ❑ 如何实施版本管理；
- ❑ 版本管理常用的工具。

### 2.1.1 版本管理的重要性

大家在进行网络维护的过程中，也许遇到过拿着一份错误的规划表对设备进行了配置，随后网络出现了故障；也许还遇到过相同角色的设备上存在着不一样的策略。例如，一对 SR（Service Router，业务路由器）和 RR（Route Reflector，路由反射器）之间建了 BGPv4 邻居。但是其中一台设备并没有发送 BGP Community（BGP 路由的一种属性），导致流量进行切换时发生了故障。这些问题都与版本管理有着密切的关系。

其实，版本管理在任何管理领域都是一项非常重要的内容。现代社会存在着大量协作，大家在互相沟通合作的时候需要确保信息的统一准确，因此信息的版本管理是非常重要的。即便是由一个人完成的工作，由于时间的关系也需要做好版本的管理工作。只要在过程中出现了信息变化，就需要有版本管理。版本管理存在着如下两个基本要素。

首先，版本管理需要有一个明确的位置来存放这些信息。在考虑了安全性之后，存放信息的位置应该是便于查询的。

其次，存放这些信息的地方需要有一个好的手段来知道它们的版本信息，既要方便知道最新的版本内容，也要方便查询历史的版本内容。图 2-1 所示就是一种非常糟糕的信息保存方法。单纯通过文件名的方式是比较难于管理的，但是这又恰恰是我们日常管理版本最常用的方法。

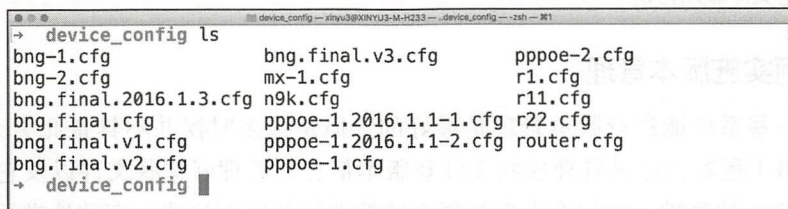


图 2-1 设备配置文件名

### 2.1.2 需要管理哪些文档

网络管理通常可以分为三个阶段：网络规划、网络建设与网络运维，每个阶段都会产生很多的信息。这些信息很多是通过文档（这里说的文档指一类文件）的形式来体现的。部分公司会采用一些软件系统进行替代，将其保存在数据库中。表 2-1 列出了一些较为常见的文档类型及其常用格式类型。



表 2-1 常见文档

阶段	文档类型	常用格式类型
规划	可行性研究	Word、Excel、PowerPoint
	测试方案	Word
	概要设计	Word、Excel、PowerPoint
	详细设计	Word、Excel
	网络规划拓扑图	PowerPoint、Visio
	资源规划	Excel
建设	资源分配表	Word、Excel
	设备详细配置	TXT
	网络拓扑图	PowerPoint、Visio
	变更方案	Word
	变更脚本	Excel、TXT
	变更操作日志	TXT
运维	应急方案	Word、Excel
	资源使用情况	Excel
	运维事件记录	Excel
	设备配置的备份	TXT

除了上述的文件外，还有一种文件是二进制文件。它们通常由其他公司提供，并不是网络管理者自己产生的。对于这样的文件，通常只需要管理其版本信息即可。

上面描述的这些文档都是需要进行版本管理的。这些文档的类型基本上是 Office 文档的格式或者纯文本的格式。

2.1.3 如何实施版本管理

如果有一些系统能进行版本管理是最好的，但是很多时候并不具备此类完善的系统，这就需要网络工程师自己来管理这些文件和版本信息。管理好这些文件以及它们的版本信息是 NetDevOps 的基础。如何在没有管理系统帮助的情况下快速而有效地进行管理呢？首先，我们需要对这些文件按照格式进行分类。这些文件大体可以分为以下三类：

- ❑ 微软 Office 格式的文件，如规划与描述的文件；
- ❑ 纯文本文件，如设备配置与一些脚本文件；
- ❑ 二进制文件，如网络设备使用的软件操作系统。这一类文件并没有包含在表 2-1 中。这是因为这种文件通常是由设备厂商提供，并不是网络管理者自己产生的。

其次，笔者建议尽量使用纯文本的格式来编写文档。Office 格式的文件通常使用文件名和文件中的版本信息来进行管理。例如，文件名为《XXX 网络实施总体方案 V1.2》。在文件的开头会有如图 2-2 所示的内容。这样的版本信息是比较常见的版本管理方法。通过阅






读文件中的版本信息，我们可以非常清晰地了解这个文件的相关版本信息。它适合在不同的人员之间、不同的部门，乃至不同的公司之间进行单个文件的传递。虽然我们可以在“文档变更过程”这里找到一些文档修改的简要信息，不过采用这样的方式很难在文件中保留全部的历史细节信息。因此，一些公司会使用微软 SharePoint (<https://products.office.com/zh-cn/sharepoint/collaboration>) 工具来进行团队的文档管理。它可以提供历史版本的管理，以及不同版本之间的比较信息。

### 版本控制信息

文档属性			
属性	内容		
项目名称	XXX 公司 2016 年 IP XX 网 XXXX 扩容 XX 期工程		
文档标题	XXX 网络实施总体方案		
文档版本号	V1.2		
版本日期	2017.3.15		
作者	张三		
审核人	李四		

文档变更过程			
版本	更新日期	更新人	主要更新内容
V1.0	2017.2.27	张三	初稿
V1.1	2017.3.2	张三	修改内容为 XXXXX
V1.2	2017.3.15	张三/李四	修改内容为 XXXXX
.....	.....	.....	.....

图 2-2 文档版本信息

 在网络管理中常常会遇到网络拓扑图信息，这样的内容是比较难以管理的。这里笔者推荐大家使用 DOT 文件格式进行网络拓扑的绘制。DOT 是一种描述图形的语言，它能用一种简单的方式来描述图形，而且能兼顾人和机器同时读取和处理。

例如，在代码清单 2-1 中描述一个简单的 IDC 的拓扑：

代码清单2-1 一个简单的拓扑

```
graph G {
    spine1 [label="核心1" color=blue]
    spine2 [label="核心2"]
    leaf1 [label="接入1"]
    leaf2 [label="接入2"]
    leaf3 [label="接入3"]
    leaf4 [label="接入4"]
    leaf5 [label="接入5"]
    leaf6 [label="接入6"]
    spine1 -- leaf1 [color=red];
    spine1 -- leaf2;
    spine1 -- leaf3;
```



```

spine1 -- leaf4;
spine1 -- leaf5;
spine1 -- leaf6;
spine2 -- leaf1 [color=red];
spine2 -- leaf2;
spine2 -- leaf3;
spine2 -- leaf4;
spine2 -- leaf5;
spine2 -- leaf6;
}

```

解析这个 DOT 文件在 Chrome 浏览器（需要安装 DOT Lang Viewer 插件）中的显示结果如图 2-3 所示。使用 DOT 语言可以借助文本的方式来保存一个网络拓扑图。<http://www.graphviz.org> 提供了 DOT 的详细说明，读者可以参考。

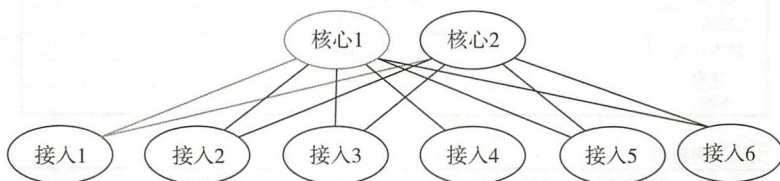


图 2-3 Chrome 浏览器显示拓扑图

再者，对于纯文本的文件，建议读者学习和了解 Markdown 的文件格式。Markdown 具有以下优点。

- ❑ 兼容性好。纯文本的文件格式兼容性非常好，任何平台下都可以完全兼容。
- ❑ 可以转化为其他格式。Markdown 可以很容易地转换为 HTML、PDF 等文件格式，并且经过了一定的排版和格式处理。
- ❑ 语法简单。Markdown 的语法非常简单，很容易学习和应用。

最后，二进制的文件相对不太好管理。建议大家最好保留每个二进制文件的 MD5 或者 SHA 的 HASH 值。Linux 和 MAC OSX 很容易对此类系统文件进行 MD5 与 SHA 值的计算。例如：

```

$ md5 iosxrv-k9-demo-6.1.2.qcow2.tgz
MD5 (iosxrv-k9-demo-6.1.2.qcow2.tgz) = 71d3be46fb68f8058b6c683f80a0f410

```

通过管理文件的 HASH 值可以确定一个文件的内容是否完整。即使文件名称被修改了，HASH 值也不会发生变化。其值不变就可以认为是相同的文件。大家熟悉的云网盘也是通过这个方法进行海量文件管理的。

### 2.1.4 版本管理的工具

在版本管理部分，存在很多的工具。目前较为流行的一个工具是 Git (<https://git-scm.com>)。Windows、Linux 和 MAC OSX 都支持 Git。Git 是一个分布式的版本控制软件，由



大名鼎鼎的 Linux 之父 Linus Torvalds 所开发，并于 2005 年以 GPL 方式发布，其最初目的是更好地管理 Linux 内核开发。现在这个工具几乎是最为流行的版本管理软件。目前 GitHub (<https://github.com>) 是一个通过 Git 进行版本控制的软件源代码托管服务中心。现在 GitHub 不单单有软件的源代码，很多软件的使用说明也通过 GitHub 进行管理。这些内容通常被放在 Git Pages (<https://github.io>) 中。对于本书的读者而言，掌握基本的 Git 工具是后续章节学习的基础。出于篇幅的考虑，这里不对 Git 的使用方法进行展开，读者可以参考 <http://rogerdudler.github.io/git-guide/index.zh.html>，这个文档是一个非常简洁的 Git 入门教材，其还包括多语言的版本。本书的后续章节中也会遇到 Git 相关命令，本书后续会默认大家已经了解和熟悉 Git 的基本命令。

## 2.2 编程语言的选择

NetDevOps 中有开发的部分，开发就必然会涉及编程语言。网络工程师在初次接触 NetDevOps 时，都会遇到编程语言的选择问题。下面我们就来简单讨论一下如何选择编程语言。这里我们将从程序语言的选择和数据描述语言的选择两个部分来进行叙述。

### 2.2.1 程序语言的选择

在程序员的世界里，讨论哪种编程语言是最好的语言，往往会引起非常激烈的争吵。笔者不敢在这里和大家讨论哪种语言是最好的语言，而是从自己的角度和大家分享一下如何选择适合 NetDevOps 的编程语言。

众所周知，编程语言既有编译型语言，也有解释型脚本语言。通常来说，编译型程序执行速度快，同等条件下对系统要求相对较低，C、C++、C# 就是这类语言。相对于编译型语言的存在方式，解释型语言的源代码不是直接翻译成机器语言，而是先翻译成中间代码，再由解释器对中间代码进行解释运行。例如，Python、Ruby、JavaScript、Perl、Shell 等就是解释型语言。

在网络运维和管理中，程序的执行效率也许不是最为关键的。换句话讲，再慢的程序（只要不出错）也会比人工执行效率要高。程序大量的执行过程是需要和网络设备进行交互的。脚本程序在执行过程中很多时候都是在等待网络设备侧返回的数据信息，因此执行效率瓶颈往往并不是语言的代码执行效率。另外，工程师也许更加关心的是程序开发的效率、代码的可读性方面。从这些方面考虑，解释型语言是首选语言。

在众多的解释型语言中，哪些语言更加合适呢？现在程序在开发的时候，通常会采用 Web 方式进行程序发布。基于 Web 的程序通常会分为前端和后端两大部分。前端开发主要解决视觉的问题，即如何更好地展现数据。在浏览器中展现数据，JavaScript 几乎是唯一的选择，当然还有 HTML 以及 CSS 等语言的方式。除了前端，后端的内容也是非常重要的。后端的主要功能是获取、整理以及保存数据。在某些时候，为了简化开发的工作量，只保





留后端的部分也是很正常的。对于 NetDevOps 网络工程师而言，如果只是为了快速地完成工作内容，将需要更多地关注后端的开发。在后端开发中，较常见的语言有 Python、Ruby、Bash、Java。其中，Python 与 Ruby 在 Web 后端开发中非常有优势。另外，NetDevOps 工程师还需要和网络设备打交道，网络设备的厂家很多。近几年，网络设备的可编程能力越来越强。基于笔者的观察，大量的网络设备会支持 Python 与 Bash 两种语言。

因此，笔者建议 NetDevOps 可以选择 Python 与 Bash。如果要兼顾前端的开发，那么也需要了解和掌握一些 Javascript、HTML、CSS 相关知识。在本书的后续章节中，使用的语言主要是 Python 与 Bash，我们会分别介绍一下这两种语言的基本语法，以及在网络运维和管理中基于这两种语言开发的常用一些工具。

## 2.2.2 数据描述语言的选择

2.2.1 节提到了编程语言的选择问题，其选择余地相对较大，并且笔者给出了认为更加适合的语言，以减少初学者在语言选择上的徘徊。但在数据描述型语言的选择方面会少很多，这里笔者给出一些常见的、在 NetDevOps 开发中通常会用到的格式。这里提到的数据描述型语言，读者最好能较好地掌握（其实掌握这些内容比学一门编程语言要简单很多）。

数据描述型语言主要是为程序提供服务的，也就是说程序能够快速方便地解析它们。部分数据描述型语言还兼顾了人的可读性，让人也能够较为方便地编写和阅读其内容。

下面是常见的数据描述型语言。

### 1. JSON

JSON (JavaScript Object Notation) 是一种轻量级的数据交换语言，以文字为基础，且易于让人阅读。JSON 数据格式与编程语言无关，它脱胎于 JavaScript，但目前很多编程语言都支持 JSON 格式数据的生成和解析。JSON 用于描述数据结构，其形式如下：

{名称: 值}

- ❑ 对象 (object)：一个对象以 “{” 开始，以 “}” 结束。一个对象包含一组非排序的名称与值对。
- ❑ 每个对内的名称与值 (collection)：名称和值之间使用 “:” 隔开。
- ❑ 多个对 (名称与值对) 之间使用 “,” 分开。
- ❑ 值的有序列表 (array)：一个或者多个值用 “,” 分区后，使用 “[]” 括起来就形成了列表。

```
{["hostname": "Router1", "hostname": "Router2"]}
```

JSON 格式表达的是一种树状的数据类型，非常容易被保存到 NoSQL 数据库中，如 MongoDB。在网络编程中，这种结构较为常见，也易于使用。比如在 Cisco Nexus 系列的交换机上就可以直接输出为 JSON 格式的数据。



```
{
  "ins_api": {
    "type": "cli_show",
    "version": "0.1",
    "sid": "eoc",
    "outputs": {
      "output": {
        "body": {
          "hostname": "switch"
        },
        "input": "show switchname",
        "msg": "Success",
        "code": "200"
      }
    }
  }
}
```

## 2. XML

XML (Extensible Markup Language, 可扩展标记语言) 是一种标记语言, 用于传送及携带数据信息, 不用于表现或展示数据。这样的结构并不太适合人直接阅读。但 XML 中的 tag 是可以携带多个属性的。并且, XML 还有 namespace 等概念。相比 JSON, XML 能表示更加复杂的数据结构, 也比 JSON 复杂很多。XML 格式是 NETCONF 协议的默认交换数据的格式。比如 Juniper 的路由器、交换机等网络设备上运行的 JUNOS 可以直接输出 XML 格式的数据。XML 是一种结构化的文本, 非常适合程序进行处理。

```
user@host> show chassis alarms
No alarms currently active
user@host> show chassis alarms | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
  <alarm-information xmlns="http://xml.juniper.net/junos/10.4R1/junos-alarm">
    <alarm-summary>
      <no-active-alarms/>
    </alarm-summary>
  </alarm-information>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

## 3. YAML

YAML (<http://yaml.org>) 是一种用来表达数据序列的格式, 对人而言可读性较高, 可以简单表达清单、散列表、标量等数据形态。它采用空白符号缩进的方式, 非常适合用来表达或编辑数据结构、配置文件、文件大纲等内容。由于 YAML 使用空白字符和分行来分隔数据, 因此它特别适合用 grep、Python、Perl、Ruby 等语言进行操作。另外, YAML 没有使用各种封闭符号, 如引号、各种括号等, 这些符号在嵌套结构中会变得复杂并且难以辨



认。YAML 的语法非常简单，主要注意文本的缩进。读者可以参考如下链接：

- ❑ <http://www.yaml.org/spec/1.2/spec.html>;
- ❑ <http://docs.ansible.com/ansible/YAMLSyntax.html>。

下面的 Ansible 的配置文件就采用了 YAML 格式进行编写。

```
---
- hosts: "core"
  connection: local
  remote_user: "admin"
  gather_facts: False
  tasks:
    - nxos_interface:
      interface: "{{ item }}"
      mode: layer3
      admin_state: up
      transport: nxapi
      host: 10.255.0.75
      username: admin
      password: cisco12345
    with_items:
      - Ethernet1/1
      - Ethernet1/2
      - Ethernet1/3
      - Ethernet1/4
```

#### 4. YANG

和前面三种语言不同，YANG 不是一种数据描述语言，而是一种数据建模语言。也就是说，它是用来定义数据的数据结构的，而不是数据的实体。通过下面这个例子可以很容易理解。

```
module acme-system {
  namespace "http://acme.example.com/system";
  prefix "acme";
  organization "ACME Inc.";
  contact "joe@acme.example.com";
  description
    "The module for entities implementing the ACME system.";
  revision 2007-11-05 {
    description "Initial revision.";
  }
  container system {
    leaf host-name {
      type string;
      description "Hostname for this system";
    }
    leaf-list domain-search {
      type string;
      description "List of domain names to search";
    }
  }
}
```





```

    }
    list interface {
        key "name";
        description "List of interfaces in the system";
        leaf name { type string; }
        leaf type { type string; }
        leaf mtu { type int32; }
    }
}
}

```

这个文件使用 YANG 定义了一个交换机的输出数据结构。system 包含 host-name、domain-search 以及 interface 三个子内容。下面的输出内容是某一台交换机真实的输出结果。

```

<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<system xmlns=" http://acme.example.com/system">
  <host-name> Swith-1 </host-name>
<domain-search> abc.com </domain-search>
<domain-search> abc.net </domain-search>
<interface>
  <name>
    <name> Eth3/1 </name>
    <type> Ethernet </type>
    <mtu> 1500 </mtu>
  </name>
  <name>
    <name> Eth3/2 </name>
    <type> Ethernet </type>
    <mtu> 1500 </mtu>
  </name>
</interface>
</system>
</data>

```

通过这个例子，我们可以看出：YANG 文件定义的是一个数据结构，而设备输出的结果使用了这个定义的数据结构，并在结构中相关部分给出了具体的值。关于 YANG 语言具体的内容，读者可以参考 <http://www.yang-central.org>。

目前，由于各厂家网络设备的输出格式都不一样，并且大量的格式还是非结构化的数据格式，这种非结构化的数据格式对程序的开发并不友好。YANG 语言是专门为网络环境而开发的语言，它对网络设备的输出数据进行抽象化并提供了一个通用的语言。Google、AT&T、Microsoft、BT 等公司共同参与了 OpenConfig (<http://openconfig.net>) 项目，这个开源的项目定义了大量和网络相关的数据结构描述文件。

关于这些数据描述语言，我们在第9章中会进行更加详细的说明，并且会给出使用 Python 语言如何处理这四种用于数据描述的格式。



## 2.3 自动化工具的选择

对于大多数没有丰富编程经验的网络工程师来说，一开始就通过编程的方式来管理网络是一个不小的挑战，也许经过几个月的学习也无法开发出一个能用的小工具，这对初学者来说是一个不小的打击。那么如何能快速地开始 NetDevOps 的相关工作呢？选择一个相对成熟的自动化工具是一个不错的选择。自动化工具有很多，这里笔者将给大家介绍四个常见的工具。

### 2.3.1 Ansible

Ansible (<https://www.ansible.com>) 是 RedHat 旗下的一个自动化运维工具平台，可以用来做系统配置管理，批量对远程主机执行操作指令。其通过 SSH 协议实现远程节点和管理节点之间的通信。因此，只要是管理员通过 SSH 登录到一台远程主机上能做的操作，Ansible 都可以做到。现在 Ansible 是一个开源的软件项目。其商业版为 Tower，商业版除了可以得到 RedHat 的支持，其还是一个有丰富 GUI 的版本。下面我们以开源的项目为主进行介绍。

Ansible 从 2.1 版本开始就集成了大量网络设备相关的模块。目前 Ansible 的最新版本是 2.3（2007 年推出）。正是这些网络设备模块的集成，使得 Ansible 在网络领域得到了广泛的应用。表 2-2 列出了 Ansible 2.3 支持的网络模块以及这些模块对应的厂商和型号。

表 2-2 Ansible 2.3 网络模块

模块名称	厂家以及型号	模块名称	厂家以及型号
A10	A10 Networks	Eos	Arista Networks
Aos	Apstra Networks	F5	F5
Asa	Cisco Systems ASA	Fortios	Fortinet
avi	AVI Networks	Ios	Cisco Systems IOS Device
Bigswitch	Big Switch Networks	Iosxr	Cisco Systems IOS-XR Device
Citrix	Citrix	Junos	Juniper Networks
Cloudengine	Huawei CE Switch	Nxos	Cisco Systems Nexus
Cumulus	Cumulus Networks	Openswitch	Open Switch
Dellos10	Dell OS 10	Ovs	Open vSwitch
Dellos6	Dell OS 6	Sros	Nokia SR
Dellos9	Dell OS 9	Vyos	Vyos

Ansible 使用的开发语言是 Python，现在 Ansible 已经同时支持 Python 2 与 Python 3。其是通过编写 YAML 文件来定义 Playbook（剧本，即任务列表）的。Ansible 不需要在设备上安装 Agent 软件，但在绝大多数的情况下，还是需要被管理的机器上有 Python 的运行环境和一些基本的 Python 模块。对于网络设备的管理，由于网络设备通常不具备 Python 的运





行环境，因此，在网络设备的模块中，被管理设备可以没有 Python 的运行环境。是否需要 Python 的运行环境，取决于具体的模块。

### 2.3.2 Puppet

Puppet (<https://www.puppet.com>) 是一个可以用于多平台环境下的集中式系统配置管理平台，能够对基础设施实现自动化的管理。Puppet 通过对整个系统中的各个节点希望达到的最终状态进行描述（检查），然后由 Puppet 执行达到目标。这种思路和过程式程序有着明显的不同。编写过程式程序需要清楚地知道每一步的执行过程并达到最终的目的。举例来说，某台交换机需要增加一个 VLAN 10。基于过程的程序流程如下：登录网络设备；进入设备的配置模式（或者再进入 VLAN 配置模式中）；向设备发送增加 VLAN 的配置；最后在网络设备上提交和保存配置。这是基于过程的一个简单的流程。如果设备已经存在需要添加的 VLAN 10，那么需要在上述的流程中加入更多的内容。对于 Puppet 而言，只需要在其配置文件中增加 VLAN 10 就可以了。Puppet 负责检查交换机上的 VLAN 信息。如果 VLAN 不存在，则会被添加。这样的好处是，让网络的管理者只需要关心最后的终态，而无须处理中间的过程。如何坚持设备的 VLAN 信息，如何添加 VLAN 的配置，是由 Puppet 内的代码来完成的。

Puppet 的开发语言是 Ruby，Puppet 需要在被管理的设备上安装一个 Agent。Puppet 的服务器和被管理设备之间使用了基于 HTTPS 的 XML RPC 协议通信。目前，已经有很多的网络厂家的设备可以安装 Puppet 的 Agent。读者可以在 <https://forge.puppet.com/> 查找自己正在使用的网络设备是否有 Agent 模块。下面给出一个 Puppet 关于增加 VLAN 10 的配置文件。

```
vlan { 'resource title':  
  name => 10  
  ensure => present  
  description => VLAN10  
  provider => Cisco  
}
```

### 2.3.3 Chef

Chef (<https://www.chef.io>) 是 Ruby 与 Erlang 开发写成的配置管理软件。Chef 相当于一个脚本管理工具，但功能要强大得多，可定制性强。Chef 将脚本命令代码化，定制时只需要修改代码，其安装过程就是执行代码的过程。

Chef 主要包括三大块：Workstation、Chef Server、Chef Client (Node)。

Workstation 通常是使用者的工作电脑，使用者在 Workstation 中创建 chef-repo，并且上传到 Chef Server，chef-repo 包括 cookbooks、recipes、roles、environment 等内容。cookbooks、recipes、roles 是 Chef 对基础设施的抽象化定义。

Chef Server 用来存储 Workstation 上传的各种资源，包括 cookbooks、roles、environments、





nodes 等。我们可以使用公有的 Server，也可以通过开源项目搭建自己的企业服务器。Chef Server 提供了非常丰富的 API，用于与 Workstation 和 nodes 传输资源和数据。Chef Server 原本由 Ruby 来实现，但后来为了保持高并发和稳定性，以及能够同时服务更多数量级的 nodes，Chef Server 内核改用了支持高并发的 Erlang 程序。

Chef Client 是安装在 Node 上的一个软件。Node 是基础设施中的一台服务器，即 Chef 管理的机器。一个 Node 可以是一台物理服务器、一台虚拟机，甚至是一台交换机或路由器。如果你想要在 Node 上部署环境，那么 Node 会与 Chef Server 进行交互以获取信息，并在 Node 上执行环境初始化操作。

### 2.3.4 SaltStack

SaltStack (<https://saltstack.com>) 也是一个配置管理系统，能够维护预定义状态的远程节点。SaltStack 的核心功能如下：

- ❑ 使命令发送到远程系统是并行的而不是串行的；
- ❑ 使用安全加密的协议；
- ❑ 使用最小、最快的网络载荷；
- ❑ 提供简单的编程接口。

SaltStack 执行程序可以为纯 Python 模块。SaltStack 执行过程中收集到的数据可以发送回 master 服务端，也可以发送到任何程序。SaltStack 可以被一个简单的 Python API 调用，或者从命令行直接调用，所以 SaltStack 可以用来执行一次性命令，也可以作为一个更大的应用程序的一个组成部分。

从结构上看，SaltStack 与 Ansible 无 Agent 的设计相反，SaltStack 在部署上可以分为 master 和 minion 两个部分，其中 master 相当于统领所有机器的总管，而 minion 则是部署在被管理机器上面的 Agent 进程。

### 2.3.5 如何选择

前面介绍了四个比较主流的自动化工具。这些工具一开始都是针对服务器而开发的，对网络设备支持的能力有限。但是，随着 NetDevOps 需求日益增长，这些自动化工具开始支持网络设备。具体选择哪一个工具可以从以下几个方面来考虑。

- ❑ 所在公司使用了哪个自动化工具。尽可能地使用公司已经成熟的自动化工具。
- ❑ 使用者熟悉哪个工具。也许你在维护网络的同时，还需要维护很多的服务器，那么用原来已熟悉的工具更好。
- ❑ 使用者或开发人员更加熟悉 Ruby 还是 Python 语言。在使用了一段时间后，也许会提出更多的需求，那么熟悉工具的开发语言对后期的二次开发会有一定的帮助。当然这也不是绝对的。例如，Chef 中使用了大量的 RESTful API，理论上可以用任何语言做二次开发。



❑ 网络设备支持的情况。这需要和设备厂家一起进行评估。

如果读者对上述的几点完全不在意的话，那么笔者推荐大家使用 Ansible 来进行网络设备的管理。理由有以下几点。

首先，Ansible 提供开源的版本，也提供商业版本。使用者可以根据自己的需求进行选择。商业版本可以获得更多的技术支持。

其次，Ansible 使用 Python 语言进行开发，配置文件使用 YAML 的文件格式。为什么笔者在 NetDevOps 中更加倾向于 Python 开发，这点可以参考 2.2 节。

再次，Ansible 使用 SSH 作为通信协议。这个协议是目前网络设备大量使用的协议。使用这个协议，不用更改现在对网络设备的管理方式。并且，Ansible 支持的网络厂家的设备类型也越来越多。而且，Ansible 可以很容易地兼容 NETCONF 的协议，因为大部分厂家的 NETCONF 协议都是基于 SSH 进行通信的。

最后，Ansible 不需要在网络设备上安装任何的 Agent。这也许是最为重要的一点。

## 2.4 网络设备的编程接口

前面我们讨论了开始 NetDevOps 之前的一些知识点，这些内容基本上还没有涉及具体的网络设备这一侧。既然是 NetDevOps，我们必然需要和网络设备打交道。网络设备需要什么能力才能更加适合 NetDevOps？我们如何看待与评估？这是本节将要阐述的内容。

### 2.4.1 网络设备接口的分类

网络设备的核心任务是转发数据包，网络工程师的核心任务是控制网络设备按照预定的设计转发数据包。网络工程师在控制网络设备的时候有以下三类方式：

- ❑ 传统的接口；
- ❑ 路由协议层面；
- ❑ 数据包层面。

#### (1) 传统的接口类型

对于网络工程师而言，最传统的方式是通过命令行的方式修改设备的配置。修改设备配置的接口有多种，最常见的是用 Telnet、SSH 或者直接用 Console 口登录到设备上进行操作。除此之外，许多厂家的设备可以通过 SNMP 的 write（写）能力对设备进行配置修改。对于支持 NETCONF 的设备，也可以通过 NETCONF 接口发送 XML 或 JSON 格式的内容对设备进行配置修改，从而达到改变设备转发路径的目的。同理，部分厂家使用 NETCONF 的方式也可以归结到这一类接口。这一类接口主要用于处理非结构化或结构化的文本信息，在后续的章节中，我们会详细讨论如何处理非结构化与结构化的文本信息。

#### (2) 通过路由协议或类路由协议的类型

这类方式发送广义的 NLRI（Network Layer Reachability Information）给设备，设





备在收到这些信息后将其转化为硬件能识别的转发表项下推送到硬件上，从而完成网络设备转发路径的变更。这一类中，较为常见的还有 PCEP (Path Computation Element Communication Protocol)、BGP、BGP-LU 等协议方式。另外，著名的 OpenFlow 协议也属于这一类。OpenFlow 发送给网络设备的流表 (flow table) 并不能算真正意义上的转发表，它和 BGP-FlowSpec 在很大程度上还是比较相似的。不过，OpenFlow 拥有更多的字段，能完成更多的事情。例如，OpenFlow 定义的字段存在一些纯控制平面使用的内容，Cookie 就是一个用于控制的标示。这一类接口主要处理的是这些广义 NLRI 信息的包结构，以及转发路径和这些包的对应关系。这部分的内容比第一类的接口类型相对复杂一些，本书主要关注的是 NetDevOps 入门的内容，因此不会涉及这一部分的开发。读者如有兴趣可以参考 ExaBGP、RYU 等开源项目。

### (3) 数据包层面的类型

第三类和前两类不一样，前两类可以归纳为：通过改变设备的转发表而影响设备的转发路径。转发表是可以通过静态配置或者路由协议计算来生成的。而第三类并不修改大部分网络设备的转发表信息，而是在数据包中添加更多的包头信息，从而实现转发路径的更改。这类方式最为典型的是 Segment Routing 和 NSH (Network Services Headers)。Segment Routing 通过 MPLS 标签堆栈来增加数据包中的转发信息，网络设备通过读取数据包头中的这些信息获取相应的转发路径。而相比 Segment Routing，NSH (参考 <https://tools.ietf.org/html/draft-ietf-sfc-nsh-12>) 携带了更加灵活的信息。由于它比 MPLS Label 更加地灵活，目前用 ASIC 来实现 NSH 的识别和转发有一定的难度。但是，NSH 非常适合在软件方式的网络设备 (如 vRouter、vSwitch 等 NFV 设备) 上来实现和完成。这里举两个例子来区分一下第三类和前两类的区别。

【例 1】运送包裹，见图 2-4。包裹在经过每一个中转站的时候是完全无法控制的，也不知道下一个中转站的情况。每一个中转站通过查询包裹的目的地址，从而知道如何往哪里投递 (转发) 它。这个过程和现在的 IP 网络非常类似，它和 IP 网一个比较大的区别是物流网络往往有较大的缓存，这些缓存就是物流的中转站。即使中转站爆仓，包裹被丢弃的概率还是非常低的。

周日	09:09:47	卖家发货
	17:44:18	南京六合大厂揽投部 收寄
	19:38:20	南京国内转运中心开拆,业务员: 186
	21:56:00	南京国内转运中心封发,发往上海处理中心,业务员: 宁通特快
周一	01:45:58	上海处理中心开拆,业务员: 091913
	05:21:08	上海处理中心 封发,发往莘庄(本部),业务员:文件40
	07:50:31	莘庄(本部) 出班
	10:03:40	莘庄(本部) 妥投,业务员: 魏立刚

图 2-4 包裹运送例子





【例2】汽车导航，见图2-5。这个例子和例1看似有点类似，但是还是有本质的区别。在这个例子中，车行驶的路径在出发前就已经规划好了（假设在出发后，司机并不改变行驶路径）。在这个规划中，每个路口都有一个提示。司机在到达这个路口时会根据这个提示进行操作，完成后这个提示就会被丢弃掉，司机需要关注的只是下一个提示。这个过程和目前的IP转发是完全不一样的。Segment Routing 和 NSH 的方式就与汽车导航的方式非常相似。以 Segment Routing 为例，数据包在进入 MPLS SR 域的边缘节点时，数据包头会被加上了一串 MPLS 的标签。这些标签就像图2-5中每一个路口的提示信息。

本书不会涉及这部分的应用开发，目前此类的应用相对还比较少，这类的开发也相对复杂，已经超出了 NetDevOps 入门的阶段。但在本书的第14章会有一个关于路径计算的案例。在此案例中，当完成路径计算后，会通过 BGP 协议给网络设备发送路由信息，达到控制路径转发的目的。

○ 人民广场
↑ 进入人民大道，行驶270米
➡ 右转，进入西藏中路，行驶370米
↰ 左转，进入延安东路，行驶360米
↑ 请直行，进入延安东路辅路，行驶170米
↰ 靠左前方行驶，进入延安东路，行驶10米
↰ 靠右前方行驶，进入世纪大道，行驶2.2公里
➡ 右转，进入银城中路，行驶280米
➡ 右转，进入花园石桥路，行驶240米
➡ 右转，进入陆家嘴环路，行驶550米
○ 进入环岛，进入丰和路，行驶70米
○ 东方明珠塔-2号门

图2-5 汽车导航

## 2.4.2 网络设备编程接口的特征

2.4.1 节中提到了网络设备接口分类的问题，那么网络编程对设备的接口有什么要求呢？这里归纳了编程接口的四个基本特征：

- ❑ 结构化数据；
- ❑ 无连接与无状态性；
- ❑ 事务性；
- ❑ 幂等性。

### 1. 结构化数据

目前网络设备输出的数据主要是面向人的显示方式（大多为 CLI 输出结果），这样格式的数据缺少一些结构化的标识。对于人来说，数据的显示和信息的分类主要是通过换行和空格符进行标识的，但这样的显示方式对于程序而言并不是很方便。程序更加容易处理使用封闭符号（各种括号与引号等）的信息嵌套格式，程序对空格与换行完全没有任何要求。对于使用换行符和空格符格式化的数据，本书暂且称它们为半结构化数据（这里指的半结构化是相对 JSON/XML 而言的）。对于结构化数据以及半结构化数据的处理方法，在本书的 Python 部分会提供详细的介绍。



## 2. 无连接与无状态性

无连接指的是每次连接后只处理一个请求。服务端处理完客户端的请求，并收到客户的确认后，立即断开连接。无状态指的是对于事务的处理没有记忆性，服务端也不知道客户端是什么状态，服务端只是根据请求发送数据。现在网络设备的 CLI 接口几乎都不具备这样的能力，CLI 方式是强交互式的方式。举例来说，如果我们需要在一台 Cisco 传统路由器上配置一个接口的 IP 地址，首先需要登录到设备上，然后输入命令 `enable` 进入 `enable` 模式，接着输入命令 `config terminal` 进入配置模式，再输入 `interface xxx` 进入接口配置模式，到这里才可以进行接口 IP 地址的配置。对设备的这几个请求必须是按顺序的，并且每次请求后都不能和设备断开连接，一旦断开了连接，必须重新开始。

NETCONF 或者 RESTful 的接口基本上实现了无连接与无状态性，如果网络设备支持的话，对于开发而言将会更加方便。

对于不具备无连接与无状态性的网络设备的开发，本书后面的章节会有讨论。对于具备无连接与无状态性的网络设备，本书的后续章节也将会有涉及。

## 3. 事务性

事务性是指在执行一个操作时，要么成功执行完成，要么根本不执行。假定现在对一台交换机有如下一个事务需要操作：在交换机的上行接口中添加一个 VLAN。对于网络工程师而言，这个事务通常可以拆解为两个小事件。首先在交换机上添加一个 VLAN ID（假定原来不存在这个 VLAN），然后在上行接口中增加这个 VLAN ID。如果这两个小事件不能组成一个事务，就有可能出现如下这样的问题：提交的上行接口名称错误，导致在上行接口上增加 VLAN ID 这个事件失败；但是第一个在交换机上创建 VLAN 这个事件先执行完成，然后又没有事务性的回滚机制，最终导致在设备上留下了这样一个多余的 VLAN ID。此类的操作或许是网络设备上出现很多“垃圾”配置的原因之一。时间久了，大量的、无效的“垃圾”配置将给运维工作带来非常大的痛苦。NETCONF（<https://tools.ietf.org/html/rfc6241>）中定义的 `candidate` 配置、`commit` 以及 `Rollback-on-Error` 的功能，将能很好地保证设备具备事务性的处理能力。

## 4. 幂等性 (idempotence)

这个概念来自于数学，现在计算机科学也借用了这个概念，其含义是指重复使用同样的参数调用同一方法时总能获得同样的结果。举例来说，下面 `permit ip any 2.2.2.2/32` 这行命令就不是一个幂等的方法。设备在这个命令前自动添加了一个序列号 20。当 `t1` 这个 `access-list` 被其他人修改后，再运行一次 `permit ip any 2.2.2.2/32` 就会得到完全不一样的效果。

```
switch(config-acl)# show ip access-list t1
IP access list t1
  10 permit ip any 1.1.1.1/32
switch(config-acl)# permit ip any 2.2.2.2/32
```





```
switch(config-acl)# show ip access-list t1
IP access list t1
  10 permit ip any 1.1.1.1/32
  20 permit ip any 2.2.2.2/32
```

这里修改了 t1 这个 access-list 后，重新运行了一次 `permit ip any 2.2.2.2/32`，就得到了完全不一样的结果。这条命令如果修改为 `20 permit ip any 2.2.2.2/32`，就会解决这个问题，不过要避免序列号 20 这个在 acl 中没有被使用，那么每次运行这条命令就可以得到完全一样的结果。

```
switch(config-acl)# show ip access-list t1
IP access list t1
  10 permit ip any 1.1.1.1/32
  30 permit ip any 3.3.3.3/32
switch(config-acl)# permit ip any 2.2.2.2/32
switch(config-acl)# show ip access-list t1
IP access list t1
  10 permit ip any 1.1.1.1/32
  30 permit ip any 3.3.3.3/32
  40 permit ip any 2.2.2.2/32
```

这个例子就是幂等性的问题，这在编程时需要注意。

对于上述四个特性，如果网络设备的 API 都能直接支持将是最好的。但是在很多情况下，往往不尽如人意，很多时候我们不得不通过传统的 CLI 方式与网络设备进行交互。当然，即使没有上述的特性，也不代表我们不能通过程序来实现一些功能，只不过我们在编写程序的时候需要处理更多的内容。

## 2.5 小结

本章主要和大家介绍了在开始 NetDevOps 之前应该了解的一些准备工作。首先，管理好自己的文档，良好的文档管理是 DevOps 的前提。其次，本书介绍了编程语言选择的问题。另外，如果读者还不具备编写程序的能力，或者所在的网络规模比较小，可以尝试先用一些开源的自动化工具来实现一些小功能，代替那些经常需要重复的工作。本书的第二篇还会介绍一些 Linux 下的常用工具，希望这些工具也能帮助大家提高工作效率。最后，本书讨论了关于网络设备编程接口的情况，希望这部分的内容能帮助大家选择合适的编程接口。

从第 3 章开始，我们将通过大量的例子和大家一起开始 NetDevOps 之旅。





## 第二篇 *Part 2*

# 基础篇

基于 NetDevOps 进行网络管理，我们需要搭建合适的工作环境和合适的开发环境，这是后续开展工作和学习的基础。因此，本篇会从如下几章来介绍如何构建我们的工作环境和使用一些常用工具。

第 3 章，介绍如何在 Linux 环境下开展日常工作，如何在 Linux 管理登录设备的会话，如何使用 Telnet 工具，如何使用 SSH 工具连接设备和建立一些 SSH 隧道。

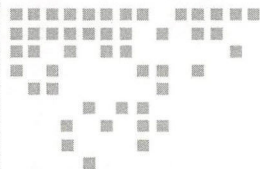
第 4 章，介绍一些 Linux 下的一些常用工具。这些工具可以帮助大家获取网络设备的一些信息，例如，通过 SNMP 协议获取网络设备的信息，获取网络的可达性信息，获取网络的 traceroute 信息。

第 5 章，介绍使用 Linux 下的常用工具来处理网络设备输出的文本内容。这里使用的工具是 Linux 文本处理的三大利器：grep、awk 和 sed。本章最后还对 vi/vim 进行了简单介绍。

第 6 章，介绍使用 Docker 来搭建 TFTP、DNS 以及 DHCP 服务器。这些服务都是网络工程师常用的一些服务。我们可以利用 Docker 来构建自行开发的工具。

希望本篇的内容能够帮助广大的网络工程在 Linux 环境下进行日常运维和开发工作。





## Chapter 3 第 3 章

# 认识命令行工具

命令行也叫 CLI (Command Line Interface), 是网络工程师最为熟悉的部分。网络工程师在管理网络设备时, 首先需要和设备之间建立通信会话。对于网络设备而言, 网络工程师使用较为广泛的有串口通信协议 (常常称为 Console)、Telnet 和 SSH。随着网络规模的增加, 网络管理者需要管理的设备数量也在不断增加, 如何更好地管理这些会话已变得越来越重要。

在当前网络环境中, 通常需要在安全性和便利性中间找到一个平衡点。当我们在管理网络设备时, 通常需要先登录到一台堡垒机, 然后通过这台堡垒机登录到具体的网络设备。这样的堡垒机通常是 Linux 平台的服务器, 它除了可以提高设备管理的安全性, 还可以让我们在服务器上部署一些网管软件、管理工具, 甚至是一些可自动化执行的脚本。

对于有一定规模的网络, 图 3-1 给出了其网络管理的抽象结构。网络管理人员不能直接登录到生产网的网络设备, 而是必须通过堡垒机登录到网络管理服务器或 Console 服务器 (Console/Terminal Server), 然后登录到具体的网络设备。在图 3-1 中, 网络管理网包含了版本管理服务器 (图 3-1 中为 Git Server, 其他版本管理软件也可以)。版本管理服务器上至少保存三个方面的内容。

- 1) 所有设备定期配置备份和每次变更后的配置。
- 2) 所有设备的操作日志。
- 3) 所有变更过程使用的脚本文件。脚本文件可以由设备配置文件组成, 也可以由代码组成或者是自动化工具的编排文件。

这样设计的原因是基于以下几个方面的考虑。

- 让所有的操作都能留下痕迹;



❑ 所有的设备配置、代码和脚本集中存放将有利于团队协作；

❑ 集中提供一个代码运行的环境；

❑ 为后续功能扩展提供逻辑空间。

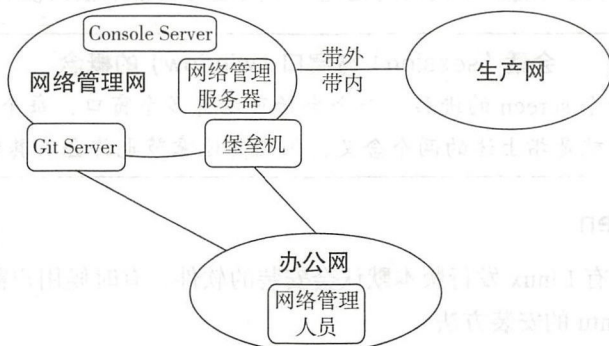


图 3-1 网络管理的抽象结构

对于小规模的网络，图 3-2 给出了更加简单的抽象结构。我们可以看到，最简化的抽象结构包含了堡垒机和版本管理服务器（图 3-2 中为 Git Server）。另外，Console 服务器则是尽可能提供，通过 Console 端口接触设备的方式是很重要的，尤其是在一些网络故障的情况下。如果在网络中实现了相对完善的 ZTP（Zero Touch Provisioning，零接触部署）和 ZTR（Zero Touch Replacement，零接触替换）环境，Console 服务器可以被省略。对于 ZTP/ZTR，本书不会涉及其细节部分，相信读者完成本书的全部内容后可以自行开发出适合自己日常应用的 ZTP/ZTR 系统。

下面我们来看看如何通过 CLI 管理网络设备。

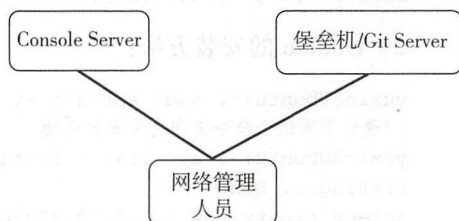


图 3-2 简化网络管理的抽象结构

### 3.1 用 screen 实现终端的会话管理

screen 是 GNU 计划开发的用于命令行终端管理的自由软件，其官方网址为 <https://www.gnu.org/software/screen/>。screen 软件有三个主要功能。

❑ 会话恢复。只要 screen 进程没有终止，其管理的会话都可以恢复。这在远程登录且网络质量不是很好的情况下非常有用，即使网络发生临时中断也可以恢复到中断前的状态。这和远程桌面（Windows RDP）是类似的，只不过这是一个文字文本界面而非图形界面。

❑ 支持多窗口。用户可以使用 screen 软件在一个会话下同时连接到多个远程虚拟终端。如果还拿远程桌面做比喻，多窗口就像一个远程桌面里面有多多个应用程序的





窗口。

- ❑ 会话共享。screen 可以允许多个用户同时登录到一个会话中，在这个会话中可以看到完全一样的输出，也可以多方同时输入。当然，这个窗口的方式是可以提供权限控制的。这样的功能在多人共同处理一个问题时会带来很大便利。

---

### 会话 (session) 与窗口 (window) 的概念

一个会话就是一个 screen 的进程。一个会话可以有多个窗口，每个窗口是一个伪终端。在本节中会话与窗口就是指上述的两个含义，但在其他章节也许会有其他的含义。

---

#### 3.1.1 安装 screen

screen 并不是所有 Linux 发行版本默认会安装的软件，有时候用户需要自己安装。这里给出 CentOS 与 Ubuntu 的安装方法。

##### 1) CentOS 的安装方法：

```
[root@Centos7 ~]# yum -y install screen
//如果你不是root用户使用如下命令
[yuxin@Centos7 ~]$ sudo yum -y install screen
//通过下面这个命令查询是否安装成功
[root@Centos7 ~]# rpm -qa screen
screen-4.1.0-0.23.20120314git3c2946.el7_2.x86_64
```

##### 2) Ubuntu 的安装方法：

```
yuxin@Ubuntu:~$ sudo apt-get -y install screen
//通过下面这个命令查询是否安装成功
yuxin@Ubuntu:~$ apt list --installed screen
Listing... Done
screen/trusty,now 4.1.0-20120320gitdb59704-9 amd64 [installed]
```

#### 3.1.2 screen 基本语法

这里列出了 screen 命令常用的参数。更加详细的应用可以参考 screen 官方手册。

screen 语法：

```
screen [-Rvx -ls] [-d <会话名称>] [-h <行数>] [-r <会话名称>] [-S <会话名称>]
```

说明如下。

- ❑ -R：先试图恢复离线的作业。若找不到离线的作业，即建立新的 screen 作业。
- ❑ -v：显示版本信息。
- x：登录到一个 screen 会话，即使这个会话已经被人使用了也可以同时登录。如果系统不只有一个会话，需要指定会话名称。在多用户环境下需要小心，可能会导致死循环。
- ❑ -ls 或 -list：显示目前所有的 screen 作业。



- ❑ `-d <会话名称>`: 将指定的 `screen` 会话离线。
- ❑ `-h <行数>`: 指定缓冲区的行数 (和内存大小有关)。
- ❑ `-r <会话名称>`: 恢复离线的 `screen` 会话。如果 `screen` 会话是一个激活的会话, 将无法再次打开。
- ❑ `-S <会话名称>`: 指定 `screen` 会话的名称。
- ❑ `-wipeⒺ`: 检查目前所有的 `screen` 会话, 并删除已经无法使用的 `screen` 会话, 用于清理一些死进程。

### 3.1.3 screen 基本操作

#### 1. 创建会话

安装完后, 直接输入 `screen` 命令就可以启动 `screen` 会话。使用这样的方式启动的 `screen` 会话是没有名字的, 不方便后续的辨认。因此, 通常启动 `screen` 会话的方式如下:

```
[root@Centos7 ~]# screen -S yuxin
```

启动 `screen` 会话后, 默认会创建第一个窗口, 这个窗口的编号是 0 (在计算机的世界里, 大部分都是从 0 开始编号的), 并且它会启动一个系统默认的 `shell`。如果你对 `screen` 没有做过任何的配置, 那么这个 `shell` 和你之前启动的 `shell` 没有什么区别。这似乎让你感觉什么也没有发生, 仿佛只是做了一个清屏的动作 (`clear` 命令)。其实, 你已经开启了一个 `screen`。在 `screen` 命令后可以直接加上你想执行的命令。例如, 你想直接登录一台设备 (见下面的命令)。当你退出登录这台设备的同时, 这个 `screen` 的这个窗口也就结束了。如果这个窗口恰巧又是这个 `screen` 的最后一个窗口, 那么整个会话也就结束了。

```
[root@Centos7 ~]# screen -S yuxin ssh admin@10.74.82.252
```

#### 2. screen 会话中的常用命令

在 `screen` 会话中, 所有的命令都是以 `C-a` (快捷键 `Ctrl+A`) 开始的。下面列出的命令形式表示为: 先按组合键 `Ctrl+A`, 松开后再按空格键。表 3-1 给出了 `screen` 常用的命令。这些命令都是系统默认的, 我们也可以根据自己的需要进行定制和修改。如何定制和修改这些命令请参考官方的文档。

表 3-1 screen 常用命令

命 令	解 释
C-a ?	显示所有键的帮助信息
C-a c	创建一个新的窗口
C-a n	切换到下一个窗口

<sup>Ⓔ</sup> 并不常用, 但可能会用到。没有出现在常用参数中。



(续)

命 令	解 释
C-a p	切换到前一个窗口
C-a 0..9	后面是任意一个数字，直接切换到编号对应的窗口。如果窗口 id 超过了 9，需要在 C-a 后输入一个单引号（即'），然后在提示下输入窗口 id 后回车。或者输入一个双引号（即"），这时会出现一个菜单以供选择。当然这两个方法在窗口 id 没有超过 9 的时候也可以使用
Ctrl+a [Space]	窗口从左到右循序切换
C-a C-a	在两个最近使用的窗口间切换
C-a x	锁住当前的窗口，需用用户密码解锁
C-a d	分离（detach）操作，暂时离开当前会话，将当前的 screen 会话（也许包含了多个窗口）放在后台执行，此时在 screen 会话中，每个窗口内运行的进程（无论是前台/后台）都在继续执行，即使退出了当前的服务器也不影响这些进程的运行。但是，如果在一个窗口中登录到了一台路由器或者是交换机，并且这台设备启用了 idle-timeout 的功能，那么这个窗口是会被终止的。如果会话就只有这么一个窗口，那么这个会话将是一个死进程
C-a z	当前的会话放到后台执行，在 shell 中执行 fg 命令则可回到 screen
C-a w	只是显示所有的窗口列表，但无法进行选择。如果需要选择使用双引号
C-a t	显示当前的时间及系统的负载
C-a k	强行关闭当前的窗口。screen 会问你是否真的要 kill 这个窗口，y 表示 yes，n 表示 no

3.1.4 定制你的 screen

screen 提供了丰富且强大的定制功能，其默认两级配置文件的位置是 /etc/screenrc 与 \$HOME/.screenrc（/etc/screenrc 是整台服务器全局的配置，\$HOME/.screenrc 是当前用户的配置；当配置冲突时，优先使用 \$HOME/.screenrc 里面的配置）。设置 screen 选项、定制命令键、设置会话启动的窗口信息与日志信息、启动多用户模式、设置用户的访问权限等，都可以在配置文件中设置。

```
[root@centos7-1 ~]# cat .screenrc
hardstatus alwayslastline
hardstatus string "%{.bW}%-w%{.gY}%n %t%{-}%+w  %=%{..G} %c:%s %M-%d-%Y"
startup_message off
vbell off
defutf8 on
```

图 3-3 是笔者常用的 screen 配置界面，screen 窗口下方始终显示窗口的名称和系统时间。这项设置是通过上面配置命令中 string 后面的参数来设定的。如果读者有兴趣定制自己的风格，可以参考 screen 的文档，或者在互联网上搜索其他人的配置样例。

3.1.5 用 screen 连接串口

网络工程师经常需要使用 console 串口连接网络设备。Apple MAC OSX 和 Linux 的



系统默认没有类似 Windows 平台下的超级终端软件，当然大家可以选择商业软件（如 SecureCRT），其实也可以使用 screen 软件来连接串口（Console 口）。



图 3-3 screen 界面

下面以 Apple MacBook Pro 计算机为例进行介绍。

首先，需要安装 USB 转串口的驱动。这个需要根据大家购买的串口转接头具体型号而定，只要厂家能提供 MAC 下的驱动就可以。驱动的安装需要根据厂家的指导来完成。

其次，需要把 USB 转串口转接头部件连接到计算机上，然后输入下面的命令用于查找刚刚连接到计算机的设备信息。这个命令用于查询 MAC OSX 计算机包含了哪些串行口的设备。

```
$ ls /dev/cu.*
/dev/cu.iirxon-DevB          /dev/cu.usbserial-FT9SI3M5
```

这里 /dev/cu.usbserial-FT9SI3M5 就是笔者的 USB 转串口设备。下面使用 screen 来打开这个串口，命令的最后一个参数是串口需要使用的波特率值。这样就可以连接到 console 设备了，命令如下：

```
$ screen /dev/cu.usbserial-FT9SI3M5 9600
```

在 Linux 环境下，/dev/ttyS0 和 /dev/ttyS1 是主板自带的串口，USB 的串口默认为 /dev/ttyUSB0 或 /dev/tty/USB1 等。在找到这些设备名后，screen 的使用方式与前面 MAC OS 的完全一致。如果读者有兴趣可以使用树莓派 (<https://www.raspberrypi.org>) 加多个 USB 转串口的部件来实现一个自制的简易 Console Server<sup>①</sup>。最新的树莓派 3 集成了 Wi-Fi 和蓝牙接口，通过它 DIY (Do It Yourself) 一个无线的 Console Server 也不是难事。

<sup>①</sup> 具体内容可以参考 <https://www.raspberrypi.org/forums/viewtopic.php?f=36&t=50735>。

### 3.1.6 管理 screen 的日志

screen 日志的记录方法有以下几种。

第一种方法：在启动 screen 的时候加上 -L 的参数。

```
[yuxin@Centos7 ~]$ screen -L -S yuxin
[yuxin@centos7-1 ~]$ ls
screenlog.0
```

第二种方法：启动 screen 的时候不加 -L 的参数。启动后使用命令 C-a H，同样会在当前目录下生成 screenlog.0 的日志文件。第一次使用 C-a H 命令，开始记录日志，屏幕左下角会显示 Creating logfile “screenlog.0”。当第二次使用 C-a H 命令后，停止记录日志，屏幕左下角会显示 Logfile “screenlog.0” closed。

这两种方式都有缺点，其文件名是固定的，不能根据会话名和时间自动创建日志文件名，这为后续的整理带来了许多不便。

第三种方法：修改配置文件。配置文件可以是 /etc/screenrc 或者是 \$HOME/.screenrc。我们在配置文件中加入如下配置：

```
logfile $HOME/log/screenlog_%S_%t_%Y_%m_%d_%c.%n.log
```

首先要确保 \$HOME 目录下有 log 文件夹。如果没有，可以通过 linux 命令——mkdir \$HOME/log 来创建。然后使用如下命令启动一个 screen。

```
[yuxin@centos7-1 ~]$ screen -L -S yuxin -t router1
[yuxin@Centos7 ~]$ ls log
screenlog_yuxin_router1_2017_05_09_14:55.0.log
```

这时我们可以看到在 log 文件夹中产生了一个 log 文件。上述配置文件中各参数含义如下：

- ❑ %S 表示会话的名称；
- ❑ %t 表示窗口的名称；
- ❑ %Y 表示年；
- ❑ %m 表示月；
- ❑ %d 表示日；
- ❑ %c 表示时间；
- ❑ %n 表示窗口 id。

### 3.1.7 多人共享一个会话

在多人协作的情况下，这是一个非常好用的功能。首先，需要开启 screen 的多用户模式。在 /etc/screenrc 配置文件中加入如下配置：

```
multiuser on
```

然后用户 A 启动一个 screen 会话，此时如果允许另外一个用户能够访问自己的会话，需要在这个会话中添加权限，方法是在 C-a 后输入 :acladd <username>。然后使用命令：

```
[yuxin@centos7-1 ~]$ screen -ls
There is a screen on:
    24631.yuxin      (Multi, attached)
1 Socket in /var/run/screen/S-yuxin.
```

另外一个用户如果需要同时登录这个会话，输入的命令如下：

```
$screen -x <username>/24631.yuxin
```

这样，两个用户可以同时看到相同的操作，无论是登录设备的方式是基于串口协议（也称为 console）还是 Telnet 或 SSH。这个功能在多人进行协作时还是很方便的。

screen 是一个会话管理软件，可以很好地管理终端会话。虽然 screen 这个软件并不大，但是其功能非常丰富。上面描述的功能只是一些常用的功能，更多的功能可以参考其手册。除了 screen，tmux (<https://tmux.github.io>) 也是具有类似功能的软件。

## 3.2 用 Telnet 和 SSH 管理设备

Telnet 与 SSH 是登录网络设备常用的工具，本节将和大家一起回顾这两个工具的使用方法。众所周知，SSH (Secure Shell) 是一个应用层和传输层上的安全协议，并包括了一些扩展功能，因此，本节的重点将是 SSH。

### 3.2.1 Telnet

Telnet 可以算是一个古老的协议，IETF 在 1983 年通过 RFC 854 发布了这个协议。RFC 854 中指出 Telnet 只能工作在 TCP 协议上，UDP 是不支持的。因为 Telnet 采用明文传送数据包，安全性不高，因此笔者强烈建议在管理网络设备时采用 SSH 方式。

Telnet 的使用非常简单，相信广大读者并不陌生，其基本命令如下：

```
$ telnet <host> <port>
```

<host> 可以是 IP 地址或者是主机名，<port> 为 TCP 的端口号。登录设备后，后续根据设备提示输入相关信息就可以了。Telnet 的命令虽然很简单，但是它也有很多参数。下面列出几个常用的参数<sup>①</sup>：

- ❑ -4：强制使用 IPv4；
- ❑ -6：强制使用 IPv6；
- ❑ -l< 用户名称 >：指定要登入远端主机的用户名称；
- ❑ -S< 服务类型 >：设置 Telnet 连线所需的 IP TOS 信息。

当 Telnet 后面为 IP 地址时，-4 -6 会没有什么意义，当 Telnet 后面为主机名时，-4 -6

<sup>①</sup> Linux、MAC 和 Windows 平台的 Telnet 命令参数可能会有一些小区别。这里主要是针对 Linux 平台。



才会有意义。很多系统默认会用当前的用户名作为远程登录的用户名，那么用 `-l` 的参数指定用户名是很有必要的；否则，第一次登录只能选择失败（某些网络设备并不会）。最后，`-S` 在某些特定情况下可以使用，例如，做 TOS 的功能性验证时，又或者你确实需要使用不同的 IP TOS 值来登录设备，达到更好的管理效果。

除了使用 Telnet 进行设备访问外，Telnet 还可以探测设备或服务器是否开放了某个 TCP 的端口。使用的方式是，直接 Telnet 设备或服务器的 IP 和 TCP 端口。如果在输入命令后，没有得到端口无法打开的信息，那么我们可以初步判断刚刚 Telnet 的端口是关闭的。在 4.2.1 节中，我们还会介绍其他功能更加丰富的工具用于端口的探测。

### 3.2.2 SSH 介绍

最初的 SSH 协议是由芬兰人塔图·于勒宁在 1995 年设计开发的。SSH 协议框架通过 RFC 4251 发布。但受版权和加密算法等限制，现在被广泛采用的软件则是 OpenSSH (<https://www.openssh.com>)。目前 OpenSSH 是 OpenBSD (<https://www.openbsd.org>) 的子项目。很多时候，OpenSSH 常被误认为 OpenSSL (<https://www.openssl.org>) 的子项，但实际上这是两个单独的项目，有着不同的开发团队。但这两个项目有着同样的目标，即提供开放源代码的加密通信软件。



**拓展** OpenSSL 是 1998 年开始的开源项目，而 OpenSSH 是 1999 年基于免费的 SSH 1.2.12 版本开发的开源项目。目前，OpenSSH 除了在加密算法的一些数学方法上使用了 OpenSSL 的开源库，其他都已经逐步抛弃了对 OpenSSL 的依赖。但是，在很多 Linux 发行版本中，OpenSSH 还是需要依赖于 OpenSSL。现在的网络设备也越来越多地基于 Linux 平台进行开发，因此在网络设备上使用的 SSH 服务端与客户端也大量地采用了 OpenSSH。

Ubuntu OpenSSH 版本信息：

```
yuxin@Ubuntu:~$ ssh -V
OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.8, OpenSSL 1.0.1f 6 Jan 2014
```

Cisco Nexus OS 7.3 OpenSSH 版本信息：

```
switch# ssh -V
OpenSSH_6.2p2, CiscoSSL 1.0.1p.4.13-fips
```

OpenSSH 实现的 SSH 功能除了包括服务端和客户端以外，还包括了一些其他的软件，如密钥的管理和分发等。

网络设备上 SSH 的实现有较大的差异，本节 SSH 的内容侧重于 Linux 平台上的 OpenSSH。

### 3.2.3 SSH 的基本使用

SSH 现在有两个不兼容的版本，分别是 V1 和 V2。OpenSSH 可以支持两个版本，并且

默认使用 V2 版本。如果要使用 V1 版本，需在参数中单独指定，其参数为 -1。目前只支持 V1 版本的网络设备已非常少见。

最常用的连接网络设备的命令如下：

```
$ ssh admin@10.74.83.166
```

这里，命令中的 admin 是登录设备的用户名。如果这里不指定用户名，那么系统就会使用当前的系统用户名作为登录设备的用户名。另外一个方式是使用 -l 的参数来指定用户名。字符 @ 后就是需要登录的设备的 IP 地址。除此之外，还有一个非常常用的参数，就是指定被登录设备的端口号。

```
ssh -l admin 10.74.83.166 -p 22
```



**拓展** 使用 username@host 与 -l username host 两个方法有什么区别？这两种方式从最后登录的效果来看是毫无区别的。但是，笔者更加推荐使用参数加值的方式（即后一种方式）。因为，在后期进行程序化处理的时候，后一种方式会更加优雅。

使用过 SSH 登录设备的读者都会知道，当我们第一次登录某台设备的时候，SSH 客户端会获取设备的公钥及其指纹信息。命令的执行结果如下：

```
$ ssh -l admin 10.74.83.166 -p 22
The authenticity of host '10.74.83.166 (10.74.83.166)' can't be established.
RSA key fingerprint is SHA256:SKerRlOHh9tQNMnnXExb7GvJvxA/+x98gx+yc+7UmZQ.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.74.83.166' (RSA) to the list of known hosts.
```

用户在输入“yes”后，这些信息默认保存在 \$HOME/.ssh/known\_hosts 文件中。以后再登录这台设备时，SSH 会检查登录的设备和第一次登录的设备是不是一致。如果不一致，则会拒绝登录。其现象如下：

```
$ ssh -l admin 10.74.83.166
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@      WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!      @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
SHA256:cbxs75fvW7ZnNu/M6NTetC+wCxCSpmEtlui77WQld+o.
Please contact your system administrator.
Add correct host key in /Users/yuxin/.ssh/known_hosts to get rid of this message.
Offending RSA key in /Users/yuxin/.ssh/known_hosts:7
RSA host key for 10.74.83.166 has changed and you have requested strict checking.
Host key verification failed.
```

在现实操作中，出现这样的问题通常有下面几种情况。

1) 被登录的设备重新安装了系统或者是进行了软件升级（软件升级大多不会有问题，



但不排除这种可能性)。

2) 设备更换了其他的 IP 地址或主机名, 并且更换后的 IP 地址或主机名之前被使用过。

3) 登录主机被恶意劫持了, SSH 这样设计的目的是确保每次都登录到相同的设备上。

生产环境使用这样的机制会更加安全, 当某台设备被恶意劫持时, 管理员通过这样的方式会发现。但如果在实验环境中, 这确实带来了很多的不便。如何解决主机指纹检查失败的问题有以下几种方法。

**第一种方法:** 打开 `$HOME/.ssh/known_hosts` 文件, 删除老的记录。也可以使用 `sed` 命令 (具体见下面的代码, 命令中的 IP 地址是希望被删除的), 关于 `Sed` 的使用方法, 第 5 章会具体介绍。如果 `$HOME/.ssh/known_hosts` 文件对主机名或 IP 地址进行了散列 (HASH), 那么这种方式就会失败。

```
$ sed -i -e '/^10.74.83.166/d' $HOME/.ssh/known_hosts
```

**第二种方法:** 使用 `ssh-keygen` 的命令。-R 的参数用于从 `known_hosts` 文件中删除所有属于 `hostname` 的指纹和公钥, 这个选项主要用于删除经过散列 (HASH) 主机的指纹和公钥。

```
yuxin@Ubuntu:~$ ssh-keygen -R 10.74.83.166
# Host 10.74.83.166 found: line 5 type RSA
/home/yuxin/.ssh/known_hosts updated.
Original contents retained as /home/yuxin/.ssh/known_hosts.old
```

**第三种方法:** 在登录时不检查指纹和公钥信息。这个方法应该是最为简单的, 但是这样做无疑降低了安全性。命令如下:

```
$ ssh -l admin -o StrictHostKeyChecking=no 10.74.83.166
```

我们可以通过 SSH 扩展工具来管理设备的指纹和公钥。第一种方法、第二种方法都是删除了设备的指纹和公钥, 在下次登录的时候再把主机的指纹和公钥保存到 `known_hosts` 文件中。如何才能通过命令自动更新设备的指纹和公钥? 自动更新指纹和公钥的命令如下:

先删除设备的指纹和公钥:

```
$ ssh-keygen -R <IP>
```

通过命令获取设备的指纹和公钥信息, 并保存到文件中:

```
$ ssh-keyscan -H [ip_address] >> ~/.ssh/known_hosts
```

通过这两个命令, 就可以更新设备的指纹和公钥信息。这种方法应该是最好的管理指纹和公钥的方法。

### 3.2.4 利用 SSH 远程执行命令

有时候我们只是想在设备上执行一条命令获取一些信息。比如, 我们想使用 `show version` 命令获取设备的版本信息。通常流程是, 我们先登录设备, 然后输入命令, 等待设



备返回结果后，再输入 `exit` 进行退出。对于 SSH 而言，我们可以不用这样做，只要在 `ssh` 命令后直接输入需在网络设备上执行的命令即可，如下所示：

```
$ ssh -l admin 10.255.0.80 show ip interface brief vrf management
Warning: Permanently added '10.255.0.80' (RSA) to the list of known hosts.
User Access Verification
admin@10.255.0.80's password:
IP Interface Status for VRF "management"(2)


| Interface | IP Address  | Interface Status             |
|-----------|-------------|------------------------------|
| mgmt0     | 10.255.0.80 | protocol-up/link-up/admin-up |


```

我们可以看到，在输入了网络设备的登录密码后，网络设备直接给出了命令执行结果，并且在执行完命令后，退出了 SSH 的登录会话。

除此之外，这些输出内容可以直接使用 Linux 下的文本处理工具进行处理（Linux 这些工具会在第 5 章进行介绍，如果读者之前没有接触过 GREP 工具，可以先跳过）。例如：

```
$ ssh -l admin 10.255.0.80 show version | grep uptime
Warning: Permanently added '10.255.0.80' (RSA) to the list of known hosts.
User Access Verification
admin@10.255.0.80's password:
Kernel uptime is 6 day(s), 4 hour(s), 25 minute(s), 33 second(s)
```

有些读者也许发现了，虽然这样做也许节省了一些时间，但是每次登录设备时都需要输入设备的密码。如果只是执行一次命令还好，但如果需要多次执行命令，这种做法还是很麻烦的。有没有什么好的办法进行优化呢？这里有一个较好的方法可以用来进一步简化操作。

OpenSSH 提供 ControlMaster 功能，在使用 ControlMaster 后，SSH 与设备之间建立一个 Master 连接，之后的所有连接都可以重复使用这一通道，也就是说不管有多少次访问请求，都只需要维护这一个 TCP/IP 连接。后续登录的连接可以不用再输入用户名和密码。

例如：首先，执行如下命令，建立一个 Master 连接。在建立这个连接的时候需要输入密码进行认证。

```
$ ssh -l admin -M -N -f \
-o ControlMaster=yes \
-o ControlPath='~/.ssh/master-%r@%h:%p' \
10.255.0.80
```

然后，后续的命令可以使用如下命令，这个时候就不需要再次输入密码。

```
$ ssh -l admin 10.255.0.80 -o ControlMaster=no \
-o ControlPath='~/.ssh/master-%r@%h:%p' \
show version
```

这样做好像还是很麻烦，每次都需要输入太多的参数。然而，上面这个例子中 `-o` 后的参数可以配置在 SSH 的配置文件中，这样就不用每次都输入。关于 SSH 的配置见 3.2.5 节的内容。

使用 SSH 远程执行命令的功能，再结合 3.1 节的内容，我们还可以很轻松地使用一条命令恢复一个远程的 screen 会话。这样的命令如何操作，请读者自己思考。我们在本章的小结中会给出例子。

通过使用这个功能，我们可以非常容易地从设备上获取信息，并且在获取信息的时候将不会受到 terminal length 的限制。当读者了解了第 5 章中有关文本处理工具的相关内容后，再结合这个功能就可以实现很多快速处理设备信息的方法。不过，这种方法并不能实现命令交互式的应用环境，比如要修改网络设备的配置。这是因为大部分的网络设备采用交互式的 CLI 来实现此类功能（有一些类型的设备通过一次传递多个命令来实现交互式操作；对于网络设备而言，笔者这里并不推荐采用这样的方式）。



**注意** ControlMaster 只支持 OpenSSH 的服务端和客户端，一些老的网络设备并不一定支持这个功能，比如 Cisco IOS 的平台就不支持，而 Cisco IOS-XR/Nexus OS、Juniper JUNOS 等平台可以支持。对于其他厂家的网络设备是否支持这个功能，读者可以自行去测试。

### 3.2.5 SSH 客户端常用配置

SSH 的配置文件可以在两个地方进行配置。第一个地方是文件 /etc/ssh/ssh\_config，这个文件管理本台主机（如 Linux Server）SSH 客户端的配置文件，也就是说无论哪个用户登录这台主机都会使用这个配置。第二个地方是每个用户的目录下，即 \$HOME/.ssh/config 这个文件。SSH 客户端配置的内容还是很丰富的，这里我们仅仅介绍一些常用的配置项。

```
1 Host *
2   ControlMaster auto
3   ControlPath ~/.ssh/master-%r@%h:%p
4   StrictHostKeyChecking no
5   UserKnownHostsFile /dev/null
6 Host router1_ios
7   HostName 10.255.0.78
8   User admin
9   Port 22
10  IdentityFile ~/.ssh/yuxin_rsa
```

第 1 行中的 Host \* 表示对所有的主机都应用的配置项目。

第 2 行和第 3 行在 3.2.4 节中使用到了。在配置文件中有了这个配置，在命令行中就不需要重复输入那么多的参数。

第 4 行内容在 3.2.3 节中使用过一次，即每次登录的时候不检查 SSH 服务端的指纹和公钥信息。

第 5 行表示把保存 SSH 服务端的指纹和公钥信息的文件指定为一个空文件，这样相当于不保存 SSH 服务端的指纹和公钥信息，这些参数同样可以被写在配置文件中。





接下来的配置文件中出现了 Host `router1_ios`，这里的 `router1_ios` 是自己定义的一个主机名，这个名字可以直接在命令中使用。对于后面的几行配置，从名字来看就可以清楚地知道他们的含义了（`IdentityFile` 指的是私钥的位置）。下面就是使用 SSH 登录这台机器的例子。在例子中，我们可以看到 SSH 软件会在配置文件中查找 `router1_ios` 的 IP 地址，而不是通过 DNS 获取。由于使用密钥登录，因此不用输入密码。在 3.2.6 节，我们就开始学习如何使用密钥进行登录和管理密钥。

```
$ ssh router1_ios
router1#
$ ssh router1_ios show version | grep uptime
router uptime is 2 days, 11 hours, 30 minutes
```

### 3.2.6 使用密钥登录设备

在上面的例子中，读者应该注意到笔者使用了密钥来登录设备。通过密钥来登录设备最直接的好处是不用再输入密码进行认证，这对于需要频繁登录设备的应用场景来说是有利的。另外，通过密钥对的方式来登录设备可以大大地提高登录的安全性。但随之带来的问题就是密钥的有效管理。

#### 1. 密钥对的产生

使用命令 `ssh-keygen` 可以产生密钥对。在产生密钥对的过程中，会提示输入一个密码（如果输入了密码而不是空着）。当这个私钥被读取的时候需要提供之前输入的密码，这样就有效地保证了密钥的安全性。读者也许会疑惑，本想取消输入密码的麻烦才选择使用密钥方式进行设备登录，但这里又要加密码岂不是反复输入密码了？暂且放下这个疑问，先来看看如何使用密钥对。一对密钥对分为私钥和公钥（参考 <https://zh.wikipedia.org/wiki/公开密钥加密>），在下面产生的密钥对中，`id_rsa` 是私钥，`id_rsa.pub` 是公钥。

```
[root@centos7-1 ~]# ssh-keygen -b 2048 -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
ae:96:47:1e:61:79:4c:a9:84:98:83:b5:ed:b4:0c:71 root@centos7-1
<略>
```

#### 2. 在设备上配置公钥

为了登录设备需要在网络设备上配置（导入）公钥。这里给出了 Cisco IOS 平台和 Juniper JUNOS 平台的配置方法，其他设备的配置请参考厂家提供的文档。

Cisco IOS 配置公钥：

```
router(config)#ip ssh pubkey-chain
```





```
router(conf-ssh-pubkey) #username admin
router(conf-ssh-pubkey-user) #key-hash ssh-rsa <public key>
router(conf-ssh-pubkey-data) #end

router#show ip ssh
SSH Enabled - version 1.99
Authentication methods:publickey,keyboard-interactive,password
Authentication Publickey Algorithms:x509v3-ssh-rsa,ssh-rsa
Hostkey Algorithms:x509v3-ssh-rsa,ssh-rsa
Encryption Algorithms:aes128-ctr,aes192-ctr,aes256-ctr,aes128-cbc,3des-cbc,aes192-cbc,aes256-cbc
MAC Algorithms:hmac-shal,hmac-shal-96
Authentication timeout: 120 secs; Authentication retries: 3
Minimum expected Diffie Hellman key size : 1024 bits
IOS Keys in SECSH format(ssh-rsa, base64 encoded): router.router.cisco.com
ssh-rsa <公钥字符串, 略>
```

### JUNOS 配置公钥:

```
[edit system login user yuxin]
user@host# set authentication load-key-file id_rsa.pub
.file.19692 | 0 KB | 0.3 kB/s | ETA: 00:00:00 | 100%
[edit system]
user@host# show
root-authentication {
ssh-rsa "<公钥字符串, 略>" ; # SECRET-DATA
}
```

### 3. 使用 ssh-agent 管理密钥

命令 `ssh-agent` 是用于管理 SSH 私钥的工具, 它为私钥提供了长时间且高速的缓存。其通过一个驻留在系统中的进程来实现这个功能。命令 `ssh-add` 可以将用户需要使用的私钥添加到由 `ssh-agent` 维护的列表中。之后 SSH 需要使用私钥登录设备时, 会优先在这里寻找私钥的高速缓存部分。命令如下:

```
$ eval 'ssh-agent'
Agent pid 53053
$ ssh-add yuxinpw_rsa
Enter passphrase for yuxinpw_rsa:
Identity added: yuxinpw_rsa (yuxinpw_rsa)
```



**注意** `eval` 后面使用的是反引号 (``), 位于键盘波浪号 (~) 下面, 实际为同一个键。对于美式键盘, 其位于数字 1 前面的那个键。

命令 `ssh-add` 添加了带有密码的私钥, 在添加的时候就会提示输入密码。以后在 SSH 中使用这个私钥的时候就不需要再一次提供密码。



**注意** 对于前面的疑问, 这里终于得到了解释, 带密码的私钥仅仅在添加到密码库时被要求输入密码, 而登录设备时将不再需要输入密码。



```
$ ssh-add -l
2048 9b:0a:d4:2c:00:fb:69:59:e4:6f:a5:ca:af:8e:cc:aa .ssh/yuxin_rsa (RSA)
1024 aa:50:a5:a4:14:74:27:db:89:14:e8:e8:c9:55:41:38 yuxinpw_rsa (RSA)
```

使用命令 `ssh-add -l` 可以查询现在 `ssh-agent` 管理了哪些私钥。

通过上述的方法可以让设备的登录过程更加安全，同时提升了登录设备的便利性。



对于一个用户，使用公钥认证的方式和使用 `tacacs+` 或 `radius` 的认证方式不能同时使用。要么使用公钥认证，要么使用密码认证（密码认证可以使用 `tacacs+` 或 `radius`）。公钥认证用于本地认证，而 `tacacs+` 或 `radius` 的认证方式用于集中认证。不过，无论通过哪种认证方式，都可以使用 SSH 进行登录。如果设备支持 `ControlMaster` 的功能，将可以简化登录过程中频繁输密码的过程。

### 3.2.7 使用 scp 进行文件传输

命令 `scp` 是 `secure copy`（安全复制）的简写，用于进行远程复制文件。它与远程设备通信的通道是和 SSH 一样的，加密方式也是类似的。其传输端口和 SSH 是一样的，默认为 `tcp 22`。

#### 1. 基本命令和参数

```
$ scp [参数] [原路径] [目标路径]
$ scp admin@router1_ios:/config/vios vios
```

远程设备的路径表示为 `username@hostname:path`，用户名和远程设备的名称或 IP 用 “@” 进行分割，这点和 SSH 登录方式是一致的。远程设备上的路径和设备名称中间用 “:” 进行分割，建议在这里写出远程设备的绝对路径。

下面列出几个常用的参数。

- ☐ `-q`: 不显示传输进度条，常用于脚本中。
- ☐ `-r`: 递归复制整个目录，这在复制多个文件时很有意义。
- ☐ `-l`: `limit`，限定用户所能使用的带宽，以 `kbit/s` 为单位。
- ☐ `-P`: `port`，这里是大写的 `P`，`port` 是指定数据传输用到的端口号。

#### 2. 常见网络设备的配置

对于服务器而言，通常来说使用 `OpenSSH` 的服务端就默认支持了 `scp` 的文件传输方式。但是对于网络设备而言，这个功能往往并不是默认开启的，需要我们进行额外的配置。具体到不同的网络设备会有一些差异，下面给出几款网络设备的例子。

Cisco IOS 设备：在配置了 SSH 登录之后，使用如下命令开启 `scp` 服务。

```
router(config)#ip scp server enable
```

Cisco Nexus 设备：同样和 IOS 类似，在配置了 SSH 登录之后，使用如下命令开启 `scp`



服务。

```
switch(config)# feature scp-server
```

Juniper JUNOS: 在 JUNOS 上只需要开启 SSH 服务就同时开启了 scp 的服务。

```
user@host# set system services ssh
```

### 3.2.8 利用 SSH 端口隧道转发功能

正如前面介绍, SSH 可以用于加密的远程登录、文件传输等功能。除此以外, SSH 还有一个非常有用的功能, 就是端口隧道转发功能。我们熟悉的 POP3、SMTP、FTP、LDAP 等协议都可以利用此功能, 通过 SSH 的加密隧道进行数据传输, 从而弥补其传输协议不加密的缺点。除了提供通道的加密功能, SSH 还能完成端口的转发功能(转换目的或源端口)。我们先看一下图 3-4。在图 3-4 中, 网络管理服务器通过交换机直连到了路由器的管理口。堡垒机和网络管理服务器之间有一层 NAT 设备, 堡垒机可以直接访问网络管理服务器, 但是网络管理服务器不能主动访问堡垒机。堡垒机上也没有路由器的管理网段路由。基于传统的登录方式, 堡垒机必须先使用 SSH 登录到网络管理服务器, 然后才能通过网络管理服务器再登录到路由器。如果要给路由器传文件, 也必须先从堡垒机拷贝到网络管理服务器, 然后拷贝到路由器上。这样的运维方式是非常烦琐的。在这种场景下, 我们可以使用 SSH 的端口隧道转发功能来简化此烦琐流程。



图 3-4 网络管理拓扑

先在堡垒机器上执行如下命令, 完成本地端口转发以及建立 SSH 隧道。

```
$ ssh -f -N -L 10080:10.255.0.80:22 network_mgt
```

说明如下。

- ❑ -f: 认证结束后在后台运行此命令, 通常和 -N 一起使用。
- ❑ -N: 不执行 shell 脚本或命令, 通常和 -f 一起使用。
- ❑ -L: 创建一个本地转发规则。将本地(这里指堡垒机)的某个端口(本例中为 10080)转发到远端指定设备(本例中为路由器的 IP: 10.255.0.80)的指定端口(例中为 SSH 的默认 TCP 22 端口)。其工作原理是: 本地机器分配一个端口, 一旦这个端口有了数据, 该规则就经过安全通道转发到远程主机的端口。从图 3-4 可以看





出，这里利用了堡垒机和网络管理服务器之间的 SSH 连接建立了一个 SSH 的隧道（IPv6 地址用另一种格式说明：port/host/hostport）。

❑ network\_mgt：连接到网络管理服务器的参数。其更加完整的形式如下：

```
-l yuxin 10.74.82.252 -p 10000 -i ~/.ssh/yuxin_rsa
```

因为在 \$HOME/.ssh/config 中有如下配置，这里只用了 network\_mgt。

```
Host network_mgt
    HostName 10.74.82.252
    Port 10000
    User yuxin
    IdentityFile ~/.ssh/yuxin_rsa
```

我们可以通过如下命令来查看这个转发策略是否已经运行。

```
# ps aux | grep -i "ssh -f"
root 29768 0.0 0.0 73940 1104 ? Ss 15:03 0:00 ssh -f -N -L 10080:10.255.0.80:22 network_mgt
```

现在我们可以直接在堡垒机上直接通过本地的 10080 端口直接登录到远端的路由器或交换机。

```
$ ssh localhost -l admin -p 10080
User Access Verification
admin@localhost's password:
Cisco NX-OS Software
Copyright (c) 2002-2016, Cisco Systems, Inc. All rights reserved.
<略>
switch#
```

也可以直接运行远端设备上的命令。

```
$ ssh localhost -l admin -p 10080 show system uptime
User Access Verification
admin@localhost's password:
System start time:      Sun May  7 09:32:40 2017
System uptime:         7 days, 3 hours, 26 minutes, 2 seconds
Kernel uptime:         7 days, 3 hours, 27 minutes, 37 seconds
```

还可以通过 scp 来传递数据。

```
$ scp -P 10080 admin@localhost:/scripts/maintenance-mode.py maintenance-mode.py
User Access Verification
admin@localhost's password:
maintenance-mode.py      100% 12KB 12.0KB/s 00:00
```

上面端口隧道转发的方法对网络设备并没有什么特殊的要求，即使网络设备不支持 SSH。所有功能都是在堡垒机和网络管理机之间完成的。

现在我们演示一下：先在网络设备上开启 Telnet 服务，然后在堡垒机上执行如下命令。

```
$ ssh -f -N -L 10083:10.255.0.80:23 network_mgt
```

通过 Telnet 登录设备。



```
$ telnet -l admin localhost 10083
Connected to localhost.
Escape character is '^]'.
Password:
Last login: Sun May 14 13:12:32 UTC 2017 from 10.255.0.77 on pts/0

Cisco NX-OS Software
Copyright (c) 2002-2016, Cisco Systems, Inc. All rights reserved.
<略>
switch#
```

在这个例子中，我们可以看到 Telnet 服务也使用了 SSH 的加密通道。

除了本地端口转发模式，SSH 还有远程端口转发模式，需要把 -L 的参数改成 -R。具体的细节希望读者自己去研究学习，这里将不再举例。

### 3.2.9 利用 SSH 做 Socket 代理

3.2.8 节提到的端口隧道转发功能是静态的端口的，本节介绍动态的端口隧道转发功能。这里，还是基于图 3-4 中的拓扑图。假设现在路由器是一台能提供 Web 服务的设备，我们可以通过浏览器来管理这台设备。这时，我们通常的做法也许会在网络管理服务器所在的网络（网段）中安装一台 Windows，然后通过远程桌面登录这台 Windows 服务器，运行 Windows 服务器上的浏览器来管理这台路由器。这样做显然是非常麻烦的，你完全可以换一种方式来实现。

首先，在堡垒机上运行如下命令：

```
# ssh -f -N -g -D 10088 network_mgt
```

查看端口服务情况：

```
# netstat -natp | grep 10088
tcp        0      0 0.0.0.0:10088      0.0.0.0:*          LISTEN      30041/ssh
tcp6       0      0 :::10088          :::*                LISTEN      30041/ssh
```

我们可以看到，在堡垒机上 TCP 10088 端口提供了一个服务。这个服务可以通过堡垒机和网络管理服务器之间的 SSH 隧道访问到路由器。这时候，我们在一台可以正常访问堡垒机 TCP 10088 端口的计算机上使用浏览器来直接访问路由器的 Web 服务，不过我们需要在浏览器上设置一个 Socket5 代理。对于 FireFox 浏览器，我们可以在连接设置中找到这个 Socket5 代理服务（见图 3-5）的其他浏览器，也可以查找其设置的位置，或者使用第三方的插件来完成相应的功能。

## 3.3 小结

本章主要介绍了 screen 和 SSH 两个工具，这两个工具主要用于管理会话和登录网络设



备。对于广大的网络工程师，SSH 应该不是一个陌生的工具。通过本章一些例子，大家可以了解和掌握 SSH 相关的一些其他功能。通过使用这些功能，相信能够给大家日常的工作带来更多的方便。



图 3-5 Firefox Socket 5 代理

下面来看一下在 3.2.4 节中留下的思考内容。

首先，使用如下命令查询其他机器的 screen 会话。

```
$ ssh -l root 172.16.6.130 screen -ls
root@172.16.6.130's password:
There are screens on:
  29645.mgt      (Multi, detached)
  23319.pts-5.centos7-1 (Multi, detached)
  22596.yuxin   (Multi, detached)
Sockets in /var/run/screen/S-root.
```

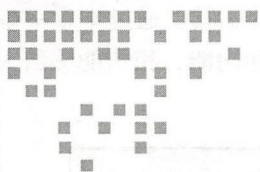
然后，使用如下命令恢复远程 screen 会话。

```
$ ssh -t -l root 172.16.6.130 screen -r mgt
```

必须添加 -t 的参数。

在这个 screen 会话中，使用 C-a d 命令分离时，会直接退出 SSH 的连接。





## Linux 下的一些常用工具

在日常网络运维工作中，分析、处理网络异常或网络故障是网络工程师基本的工作内容。如何快速定位和排除网络中的故障点是网络运维工程师体现自身价值的努力方向。为了能有效地定位问题，快速收集网络相关的数据是基础之一。本章主要介绍在 Linux 平台下的一些命令行工具。为什么是 Linux 平台？正如第 3 章介绍的，网络运维的工作环境是堡垒机或网络管理服务器，而它们往往是 Linux 平台。如果通过命令行能获取网络设备的信息，后续再加上数据分析和逻辑处理就可以实现小规模的自动化工具。图形化的展现设备运行状态信息固然非常直观，但这样的展现方式只适合人来观察。图形化的展现并不适合机器对这些数据进行二次处理。本章会从如下几个方面对 Linux 下的常用工具进行介绍。

### （1）获取网络设备运行状态信息

获取并展现网络设备的运行状态是传统网管平台的基本功能。网络告警功能也常常是通过获得这些数据而产生的。因此，本章会首先介绍通过 SNMP 如何获取设备信息的工具。这里介绍的工具为 SNMP 工具集。

### （2）网络的可达性检测

网络的基本目的是完成可达性，因此获取网络的可达性数据是非常关键的。可达性数据可以分为“通”与“不通”两种情况。这里的工具会介绍 Nmap 与 Nping。对于“通”的部分，我们又常常希望知道其更多的信息，如 RTT（Round Trip Time，往返时间）、带宽、抖动等信息。这里的工具会介绍 iPerf 与 Fping。

### （3）网络转发路径信息

网络通常是由很多节点组成的。出于安全性和稳定性的考虑，在网络设计的时候通常都会考虑冗余链路和冗余节点。因此，网络中会出现多条等价或者是不等价的转发路径。如何获取两点之间的转发路径也是很需要的。这里的工具会介绍 MTR。

除了上述这些工具之外，我们还会介绍几个常用的工具，分别是 Watch、Wget、CURL。

## 4.1 SNMP

### 4.1.1 SNMP 简介

SNMP (Simple Network Management Protocol, 简单网络管理协议) 开发于 20 世纪 90 年代早期, 其目的是在大型网络中简化对设备的管理和数据的获取。在实际的应用场景中, 这个协议在数据获取方面有非常广泛的应用。以至于在网管软件中, 一提到获取设备的数据就必定会用到 SNMP。目前 SNMP 有三个版本, 分别是 V1、V2 和 V3。V1 是一个初始的标准, 其定义了 SNMP 的基本框架。其主要缺点是难以实现大量的数据传输以及缺少身份验证和加密机制。V2 基于 V1 框架进行了修改, 目前修订为 V2c。V2c 添加了几个新的数据类型 (Counter32、Counter64、Gauge32、UInteger32 以及 BitString), 增强了对 OID 表和 OID 值的设置。但是, V2 的增强并没有完全实现预期目标, 特别是安全性没有得到提高, 仍然使用明文进行数据传输和认证。V3 体系结构引入了 USM (User-based Security Model, 基于用户的安全模型) 和 VACM (View-based Access Control Model, 基于视图的访问控制模型), 其中 USM 用于消息安全, VACM 用于访问控制。通过简明的方式实现了加密和认证功能。因此, 大家在进行网络管理时尽可能地采用 V3 版本。

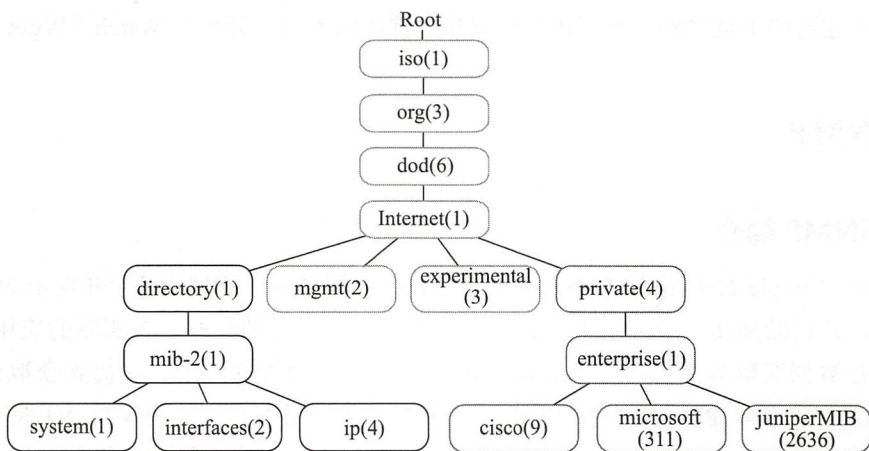
SNMP 是一个非常好的结构化数据。它的数据结构表现为 Key-Value 的形式, 虽然, 在 OID (Object Identifiers, 对象标识符) 的定义上采用树形的结构方式 (见图 4-1), 但我们在使用时通常采用二维的表结构。OID 可以看作 Key, 也就是关键字 (或者也可以称为字段)。通过这些 OID 能从设备上获取一个值。有一些 OID 还有写的的能力, 也就是说可以通过 SNMP 中的 OID 来修改设备上的值。比如, 常用的有通过 SNMP 进行 ping 测试, 通过 SNMP 修改设备的 ACL 信息等。但是, 当前对于 SNMP 的应用主要集中在读取设备的信息。通过 SNMP 获取的结构化数据比通过 CLI 输出的半结构化文本要容易处理很多 (第 11 章会介绍如何处理 CLI 的半结构化文本)。我们还可以结合第 5 章提供的一些工具, 快速地获取想要的信息。

另外, 随着云计算的发展, 对于网络监控的精度, 分钟级已经不能满足需求了。这时, SNMP 协议在高精度的监控中表现得捉襟见肘。我们可以从“推”和“拉”这两个方面来看。

首先, SNMP 在采集设备信息的时候采用“拉”的方式获取设备的数据 (SNMP trap 通常不用于设备的信息获取)。通常在服务器上会启动定时任务, 这些任务的时间间隔一般是 1~5 分钟, 但是太过于频繁地拉取设备的信息会带来以下几个问题。

①可能会导致设备控制平面的 CPU 过于繁忙, 从而使其他进程不能得到有效的资源, 最后影响设备的正常使用。





OID Tree Example

图 4-1 OID 树形结构

②设备根本无法支持过于频繁的数据采集，直接表现就是一段时间的数据是没有变化。因为在设备内部，很多数值是采用滑动平均来给出值。滑动平均会削平那些剧烈变化的数据。

③任务始终执行不完。由于 SNMP 采集服务器通常为集中式部署，任务间隔短会使执行中的任务太多。通常的表现是上一个任务还没有执行完，后一个任务就已经开始启动。

其次，“推”的方式是不是可以获得更好的效率呢？目前很多厂家逐渐开始提供更多的 telemetry(遥测) 接口。这种接口大部分采用了 GPB(Google Protocol Buffer<sup>①</sup>) 的数据格式。这是一种轻便高效的结构化数据存储格式，可以用于结构化数据的序列化。其通信协议通常采用 gRPC (Google Remote Process Call<sup>②</sup>)。另外，其采用网络设备往外“推”的方式。关于 GPB 以及 telemetry 的详细内容，本书并不会涉及。在这里提到它，是因为在高精度、高频率地采集数据时，其可以作为 SNMP 的替代品。Cisco 和 Juniper 都提供了 telemetry 的开源项目。当然，还有一些其他的开源项目。

❑ Cisco: <https://github.com/cisco/bigmuddy-network-telemetry-collector>;

❑ Juniper: <https://github.com/Juniper/open-nti>。

最后需要说明的是，虽然有更好的解决方案来代替 SNMP，但是，并不是说我们现在就可以完全放弃 SNMP 了。打个比喻来说，有了汽车并不代表我们就要完全抛弃自行车。

## 4.1.2 常见设备的 SNMP 配置

在使用 SNMP 客户端工具之前，我们需要在网络设备上先启用 SNMP 服务端的配置。

① <https://developers.google.com/protocol-buffers>。

② <http://www.grpc.io>。



为了减少读者查其他文档的时间以及便于后续 SNMP 命令的演示，这里简单地给出 Cisco IOS、Cisco IOS-XR 以及 Juniper JUNOS 的设备配置。V1 和 V2 的配置区别不大，下面分别给出 V2 与 V3 的配置。这些配置并不包含全部的 SNMP 相关内容，只提供最简单的配置。

#### (1) Cisco IOS V2

```
//最简化配置
Router#configure terminal
Router(config)#snmp-server community NetDevOps ro 1
Router(config)#access-list 1 permit host 172.16.1.78
```

#### (2) Cisco IOS V3

```
Router#configure terminal
Router(config)#access-list 1 permit host 172.16.1.78
Router(config)# snmp-server group admins v3 auth access 1
Router(config)# snmp-server group admins v3 priv access 1
Router(config)# snmp-server user yuxin admins v3 auth sha hello123 priv des priv1234
```

#### (3) Cisco IOS-XR V2

```
//最简化配置
RP/0/0/CPU0:iosxrv-1(config)# snmp-server community NetDevOps ro IPv4 acl_snmp
RP/0/0/CPU0:iosxrv-1(config)# ipv4 access-list acl_snmp permit host 172.16.1.78
RP/0/0/CPU0:iosxrv-1(config)# commit
```

#### (4) Cisco IOS-XR V3

```
snmp-server user yuxin NetDevOps v3 auth sha encrypted 130D121E0703557878 priv
des56 encrypted 105E1B101346405858
snmp-server view view_oid 1.3.6.1.2.1 included
snmp-server group NetDevOps v3 priv read view_oid
```

#### (5) Juniper JUNOS V2

```
set snmp community NetDevOps authorization read-only
set snmp community NetDevOps clients 172.20.0.0/16
```

#### (6) Juniper JUNOS V3

```
v3 {
    usm {
        local-engine {
            user yuxin {
                authentication-sha {
                    authentication-key
                        <略key>; ## SECRET-DATA
                }
                privacy-des {
                    privacy-key
                        <略key>; ## SECRET-DATA
                }
            }
        }
    }
}
```

```

    }
}
}
vacm {
    security-to-group {
        security-model usm {
            security-name yuxin {
                group NetDevOps;
            }
        }
    }
    access {
        group NetDevOps {
            default-context-prefix {
                security-model usm {
                    security-level privacy {
                        read-view view_oid;
                    }
                }
            }
        }
    }
}
}
}
view view_oid {
    oid 1.3.6.1.4.1.2636.3 include;
    oid 1.3.6.1.2.1;
}
}

```

### 4.1.3 SNMP 工具

在 Linux 平台下，使用最为广泛的 SNMP 工具集是 net-snmp (<http://www.net-snmp.org>)。它是一个开源的项目，支持 SNMP V1/V2/V3 版本，并且是一个支持基于 IPv6 的应用程序。它包括客户端工具集和服务端，这里我们主要使用客户端工具集。

下面我们来介绍几个常用的命令。

#### 1. 命令 snmpget

语法如下。

```
$ snmpget [命令选项] 网络设备 OID
```

常用命令选项如下。

- ❑ -v: 1|2c|3, 指定 SNMP 的版本信息;
- ❑ -c: community 值 (注意区分大小写)。

例如:

```
$ snmpget -v 2c -c NetDevOps 192.168.0.4 .1.3.6.1.2.1.2.2.1.2.1
iso.3.6.1.2.1.2.2.1.2.1 = STRING: "GigabitEthernet0/0"
```



在 `snmpget` 后必须使用完整的 OID 值，因为这个命令是用于获取一个 OID 值的工具。上面的例子中是获取了一个接口的名称。

## 2. 命令 `snmpwalk/snmpbulkwalk`

命令 `snmpwalk` 和 `snmpbulkwalk` 都可以获取一个树形节点下所有的值，它利用了 `GetNextRequest` 对给定的树进行遍历，一般用来对表格类型管理信息进行遍历，也可以用于对一个 OID 的值后面几位尚不确定的时候。命令 `snmpwalk` 每次和设备交互时，只能获取一个具体的 OID 值。但是，`snmpbulkwalk` 可以一次获得多个 OID 的值。因此，后一个工具的执行效率会更高。不过，`snmpbulkwalk` 需要 V2 和 V3 的版本才可以支持。如果你的 SNMP Server 只支持 V1，那只能使用 `snmpwalk` 了。

```
$ snmpwalk -v 3 -A hello123 -a sha -l authPriv -x des -X priv1234 -u yuxin
192.168.0.4 1.3.6.1.2.1.2.2.1.2
iso.3.6.1.2.1.2.2.1.2.1 = STRING: "GigabitEthernet0/0"
iso.3.6.1.2.1.2.2.1.2.2 = STRING: "GigabitEthernet0/1"
iso.3.6.1.2.1.2.2.1.2.3 = STRING: "GigabitEthernet0/2"
iso.3.6.1.2.1.2.2.1.2.4 = STRING: "GigabitEthernet0/3"
iso.3.6.1.2.1.2.2.1.2.5 = STRING: "Null0"
iso.3.6.1.2.1.2.2.1.2.6 = STRING: "Loopback0"
iso.3.6.1.2.1.2.2.1.2.7 = STRING: "Loopback1"
```

说明如下。

命令的参数和参数后面的值都是区分大小写的。

- ❑ `-a PROTOCOL`：指定认证时使用的协议为 `md5` 或者 `sha`（本例中为 `sha`）。
- ❑ `-A PASSPHRASE`：认证时使用的密码。本例中为 `hello123`。
- ❑ `-l LEVEL`：认证和加密的方式。包括三种：`noAuthNoPriv`、`authNoPriv`、`authPriv`。
- ❑ `-u USER-NAME`：认证时的用户名。本例中为 `yuxin`。
- ❑ `-x PROTOCOL`：加密方式为 `DES` 或 `AES`。
- ❑ `-X PASSPHRASE`：加密时用的密码。本例中为 `priv1234`，这个密码的认证密码可以不一样。

这里给出一个 `snmpwalk` 与 `snmpbulkwalk` 执行效率的比较，实现对一台设备进行较大数据量的获取。命令中关于非 SNMP 的部分大家可以忽略。

```
yuxin@server-1:~$ time snmpbulkwalk -v 3 -A hello123 -a sha -l authPriv -x des
-X priv1234 -u yuxin 192.168.0.4 1.3.6.1.2.1 > /dev/null
real    0m6.973s # 一共执行了6.973秒
user    0m0.104s
sys     0m0.040s
yuxin@server-1:~$
```



```
yuxin@server-1:~$ time snmpwalk -v 3 -A hello123 -a sha -l authPriv -x des -X
priv1234 -u yuxin 192.168.0.4 1.3.6.1.2.1 > /dev/null
real    0m44.714s # 一共执行了44.714秒
user    0m0.494s
sys     0m0.570s
```

通过上面的两个例子，我们可以看到 `snmpbulkwalk` 的执行效率更高。

### 3. 命令 `snmpdelta`

这个命令可以用来监视数值类型的 OID，它会及时报告值的改变情况。比如，我们获取一个接口的转发流量值。

```
$ snmpdelta -v 3 -A hello123 -a sha -l authPriv -x des -X priv1234 -u yuxin -Cs
-Cp 5 -Cm 192.168.0.4 iso.3.6.1.2.1.2.2.1.16.1
[15:23:25 5/19] iso.3.6.1.2.1.2.2.1.16.1 /5 sec: 595 (Max: 595)
[15:23:30 5/19] iso.3.6.1.2.1.2.2.1.16.1 /5 sec: 331 (Max: 595)
[15:23:35 5/19] iso.3.6.1.2.1.2.2.1.16.1 /5 sec: 241 (Max: 595)
[15:23:40 5/19] iso.3.6.1.2.1.2.2.1.16.1 /5 sec: 271 (Max: 595)
[15:23:45 5/19] iso.3.6.1.2.1.2.2.1.16.1 /5 sec: 3649 (Max: 3649)
[15:23:50 5/19] iso.3.6.1.2.1.2.2.1.16.1 /5 sec: 3129 (Max: 3649)
```

说明如下。

- ❑ `-Cs`: 输出时间戳；
- ❑ `-Cp`: 每次轮询的时间间隔（本例中为 5 秒）；
- ❑ `-Cm`: 输出最大值。

SNMP 确实是一个非常常用的工具，我们在使用它的时候，通常使用那些已经集成在网管平台中的图形化界面。SNMP 的工具集还包含了 MIB 文件的读取和查询功能，读者可以在 <http://www.net-snmp.org/wiki/index.php/TUT:snmptranslate> 查看 `snmptranslate` 这个工具。既然，本书的读者是为了在 NetDevOps 领域里有所收获，那么首先通过命令行工具获取一些数据可以作为 NetDevOps 的开始。这也是为什么本书会提供很多命令行工具。其实，很多工具都有很丰富的功能。由于篇幅的问题，本书不可能把每个工具的所有功能都介绍全面。读者可以根据本书提供的参考链接进行扩展了解。

## 4.2 网络可达性检测工具

在日常的网络维护工作中，经常需要进行网络可达性检测。网络可达性检测是最靠近业务可用性的一种测试方式。网络可达性检测通常包含如下两种层次。

### (1) 可达性检测

这是网络可达性检测的第一个层次，即要确定上层应用是否能“通”。最常见的方式是通过 ICMP 来检测。为了更加贴近应用，也会检测 UDP 或者是 TCP 的端口是否能正常访问，并且需要得到一些简单的统计值。



## (2) 测量任意两点或者是多点之间网络质量情况

在检测了可达性后，通常还需要知道连通后的一些数据，也常被称为网络质量。比如，知道两点间的传输延时、丢包率以及其他信息。这种测试在网络设备上也经常会被部署，从而获得相应的数据，比如在 Cisco 平台上称为 SLA (Service Level Agreements)，在 Juniper 平台称为 RPM (Real-time Performance Monitoring)。网络工程师对这两个内容应该不会觉得陌生。那么在 Linux 平台下如何也能获得类似的数据呢。这是本节要介绍的工

### 4.2.1 Nmap

#### 1. Nmap 简介

Nmap (<https://nmap.org>) 是一个开源的网络探测和安全审核的工具，其全名为 Network Mapper。这款软件的功能是能够快速地进行大型网络扫描。Nmap 除了提供命令行的工具之外，还提供一个图形化的操作工具，其子项目的名称是 Zenmap，界面如图 4-2 所示。

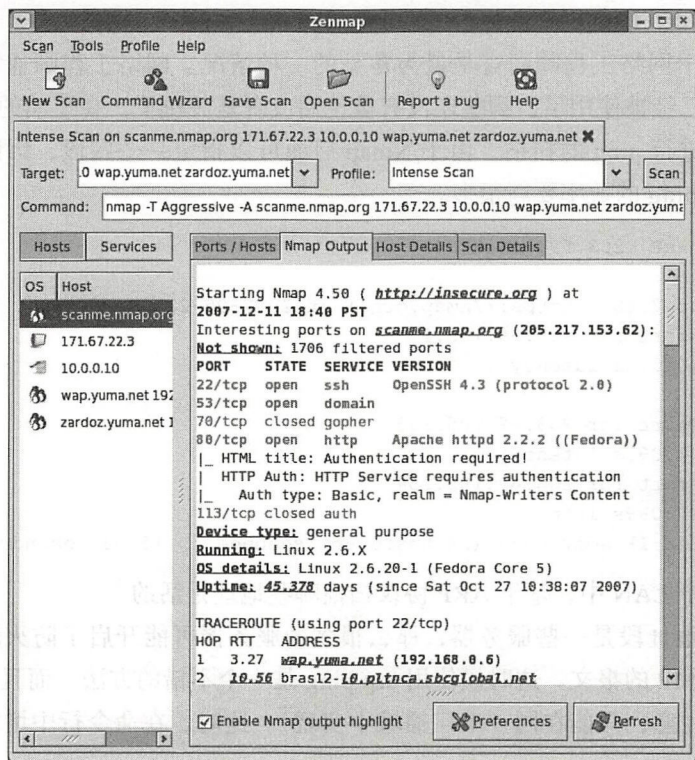


图 4-2 Zenmap (来自 <https://nmap.org/zenmap>)

其下载页面为 <https://nmap.org/zenmap/download.html>，这里提供了 Windows、MAC 以及 Linux 三个版本。这个图形化工具不仅提供了图形化的显示，对于在图形化中选择的任





何操作都会给出一个命令行的表示方式。这对于初次使用 Nmap 的人来说也是一个很好的工具。虽然，Nmap 提供的是安全审核的功能，但是系统管理员和网络管理员可以用它来辅助处理日常工作，如监控服务器的服务端口、检测网络的可达性等。

这里是命令的一个例子，其功能是扫描主机 `www.taobao.com` 的 80 和 443 端口是否打开。关于命令的具体解释，会在下面进行详细的描述。

```
$ sudo nmap -sS -p80,443 www.taobao.com
Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-20 23:04 CST
Nmap scan report for www.taobao.com (66.198.24.243)
Host is up (0.070s latency).
PORT      STATE SERVICE
80/tcp    open  http
443/tcp   open  https
Nmap done: 1 IP address (1 host up) scanned in 0.55 seconds
```

## 2. Nmap 常用参数

基于网络工程师的日常工作场景，本书将介绍如下几种场景。

### (1) 基于 ICMP 协议扫描哪些地址是可用的

这个场景对于网络工程师而言是最为常见的一种情况。网络工程师通常需要知道某个网段中哪些地址已经被使用了，哪些还没有被使用或者是关机了。在下面的命令中（注意大小写），`-sP` 是指进行 ping 的扫描。由于 Nmap 后面可以指定一个网段，因此这样比一个地址一个地址逐个 ping 的效率要高很多。

```
$ sudo nmap -sP 203.69.105.0/24
Password:
Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-21 09:20 CST
Nmap scan report for 203.69.105.1
Host is up (0.071s latency).
<中间信息略>
Nmap scan report for 203.69.105.253
Host is up (0.093s latency).
Nmap scan report for 203.69.105.254
Host is up (0.089s latency).
Nmap done: 256 IP addresses (58 hosts up) scanned in 15.53 seconds
```

### (2) 在同一个 LAN 中，基于 ARP 协议扫描哪些地址是活的

如果扫描的地址段是一些服务器，那么很多的服务器可能开启了防火墙，这些防火墙导致无法响应 ICMP 的报文。这时候使用 arp ping 是一个不错的方法。而且，这样的扫描方式也会比较快（注意，只能在同一个广播域中实现）。这时，在命令行中增加“`-PR`”参数就可以了。

```
$ sudo nmap -sP -PR 192.168.1.0/24
Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-21 09:25 CST
Nmap scan report for 192.168.1.1
Host is up (0.0035s latency).
```





```

MAC Address: D0:0F:6D: [REDACTED] (T&W Electronics Company)
<略>
Nmap scan report for 192.168.1.201
Host is up (0.021s latency).
MAC Address: 00:11:32: [REDACTED] (Synology Incorporated)
Nmap scan report for 192.168.1.3
Host is up.
Nmap done: 256 IP addresses (8 hosts up) scanned in 47.26 seconds

```

### (3) 基于 UDP 协议扫描

Nmap 工具是从一个端口扫描器发展起来的，因此端口扫描可以认为是它最核心的功能了。对于端口的扫描结果，Nmap 提供了六种结果，如表 4-1 所示。

表 4-1 Nmap 识别的端口状态

状态名称	中 文	解 释
Open	开放	有应用程序在这个端口接受 UDP 报文和处理 TCP 连接
Close	关闭	端口是关闭的，但 Nmap 可以访问。没有应用程序在这个端口监听数据。通常有应用负载均衡设备也会出现这个情况
Filtered	被过滤	Nmap 无法确定端口是否开放
Unfiltered	没有被过滤	Nmap 不能确定它是开放的还是关闭的。只有用于映射防火墙规则集的 ACK 扫描才会把端口分类到这种状态
Open filtered	开放或被过滤	Nmap 不能确定它是开放的还是被过滤的。例如，一个开放的端口没有响应。在 UDP 与 IP 扫描中较为常见
closed filtered	关闭或被过滤	Nmap 不能确定它是开放的还是关闭的（少见，忽略）

说明如下。

❑ -sU：指定为 UDP 扫描方式；

❑ -p：指定需要扫描哪个或哪些端口。

多个端口需要使用逗号隔开，使用示例如下：

```

$ sudo nmap -sU -p 53 114.114.114.114
Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-21 10:00 CST
Nmap scan report for public1.114dns.com (114.114.114.114)
Host is up (0.015s latency).
PORT      STATE      SERVICE
53/udp    closed    domain
Nmap done: 1 IP address (1 host up) scanned in 0.39 seconds

```

### (4) 基于 TCP 协议扫描

说明如下。

❑ -sT：指定为 TCP 扫描方式；

❑ -p：同 UDP，最后也可以使用主机名。

```
sudo nmap -sT -p 80,443 www.baidu.com
```



```
Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-21 10:04 CST
Nmap scan report for www.baidu.com (115.239.210.27)
Host is up (0.015s latency).
Other addresses for www.baidu.com (not scanned): 115.239.211.112
PORT      STATE SERVICE
80/tcp    open  http
443/tcp   open  https
```

对于更多扫描方式，读者可以参考 <https://nmap.org/man/zh/man-port-scanning-techniques.html>。关于更多关于 nmap 例子和用法，可以参考其官网的中文参考指南 <https://nmap.org/man/zh/index.html>。相信通过这个工具可以提高我们的工作效率。

## 4.2.2 Nping

Nping 是 Nmap 工具集中的一个工具。相信大家对 ping 的工具并不陌生，ping 的工具通过 ICMP 发送一个 ICMP echo 请求包，并等待 ICMP 的 echo 回应包，程序会按时间记录丢包率和网络延时（数据包的往返时间）。Nping 不仅仅可以使用 ICMP，还可以使用 UDP、TCP 以及 ARP 进行探测，在探测的时候还可以对一个网段进行大规模的扫描。下面是一个 TCP ping 的简单例子：

```
sudo nping -c1 --tcp -p 80 103.235.46.39/31
Starting Nping 0.7.40 ( https://nmap.org/nping ) at 2017-05-21 10:48 CST
SENT (0.0057s) TCP 10.125.15.212:29725 > 103.235.46.38:80 S ttl=64 id=12089
  iplen=40 seq=4146500087 win=1480
RCVD (0.0500s) TCP 103.235.46.38:80 > 10.125.15.212:29725 SA ttl=52 id=47335
  iplen=44 seq=2067308546 win=65535 <mss 1260>
SENT (1.0091s) TCP 10.125.15.212:29725 > 103.235.46.39:80 S ttl=64 id=12089
  iplen=40 seq=4146500087 win=1480
RCVD (1.0587s) TCP 103.235.46.39:80 > 10.125.15.212:29725 SA ttl=53 id=48588
  iplen=44 seq=185480458 win=65535 <mss 1260>
Statistics for host 103.235.46.38:
  | Probes Sent: 1 | Rcvd: 1 | Lost: 0 (0.00%)
  | _ Max rtt: 44.422ms | Min rtt: 44.422ms | Avg rtt: 44.422ms
Statistics for host 103.235.46.39:
  | Probes Sent: 1 | Rcvd: 1 | Lost: 0 (0.00%)
  | _ Max rtt: 49.476ms | Min rtt: 49.476ms | Avg rtt: 49.476ms
Raw packets sent: 2 (80B) | Rcvd: 2 (92B) | Lost: 0 (0.00%)
Nping done: 2 IP addresses pinged in 1.06 seconds
```

说明如下。

- ❑ **-c1**：表示只发送一次。
- ❑ **--tcp**：表示探测的方式为 TCP 协议。
- ❑ **-p**：指定端口号，这里和 Nmap 类似，也可以指定多个端口。每个端口都会发送一次。**-p** 后是需要测试的网段，工具会自动计算 IP 地址所在的子网信息。

在命令执行结束后有一个简单的报告，和 ping 命令类似，会给出一些统计结果。我们可以看到，这个工具对于批量进行 ping 操作是非常方便的。更多参数和功能可以参考 <https://nmap.org/book/nping-man.html>。



### 4.2.3 iPerf

iPerf 工具可以算是一个较老的工具了, 1999 年由美国伊利诺伊大学开发, 现在的版本已经是 V3, 即 iPerf3 (<http://software.es.net/iperf>, 这里可以下载软件)。正如其名, 这个工具主要的功能是测试网络的性能。网络性能的一个重要的指标是网络的吞吐量。这个工具主要用于测试 UDP 和 TCP 这两种协议的吞吐量。当大家没有专门的网络测试仪 (如 Spriant 公司的 TestCenter 或 IXIA 公司的 IxNetwork 等) 时, 用这个软件可以进行网络吞吐量的测试 (软件自身能产生的流量和服务器的性能有一定的关系)。这款软件 TCP 协议无法提供抖动的值, UDP 则可以。

在进行带宽测试时, 最少需要两台服务器, 一台作为服务端, 另一台作为客户端。服务端的命令很简单:

```
$ iperf3 -p 5000 -s -D
```

说明如下。

- -p: 指定一个端口;
- -s: 说明是服务端;
- -D: 在服务器后台运行。

客户端命令如下。

1) 通过 TCP 测量带宽。

```
$ iperf3 -c 172.16.6.130 -p 5000 -b 3G -t3 --get-server-output
Connecting to host 172.16.6.130, port 5000
[ 4] local 172.16.6.1 port 54177 connected to 172.16.6.130 port 5000
[ ID] Interval      Transfer    Bandwidth
[ 4]   0.00-1.00    sec      221 MBytes  1.85 Gbits/sec
[ 4]   1.00-2.00    sec      247 MBytes  2.07 Gbits/sec
[ 4]   2.00-3.00    sec      262 MBytes  2.20 Gbits/sec
- - - - -
[ ID] Interval      Transfer    Bandwidth
[ 4]   0.00-3.00    sec      730 MBytes  2.04 Gbits/sec      sender
[ 4]   0.00-3.00    sec      729 MBytes  2.04 Gbits/sec      receiver
Server output:
Accepted connection from 172.16.6.1, port 54176
[ 5] local 172.16.6.130 port 5000 connected to 172.16.6.1 port 54177
[ ID] Interval      Transfer    Bandwidth
[ 5]   0.00-1.00    sec      203 MBytes  1.70 Gbits/sec
[ 5]   1.00-2.00    sec      225 MBytes  1.88 Gbits/sec
[ 5]   2.00-3.00    sec      241 MBytes  2.02 Gbits/sec
[ 5]   3.00-3.25    sec      61.0 MBytes  2.03 Gbits/sec
- - - - -
[ ID] Interval      Transfer    Bandwidth
[ 5]   0.00-3.25    sec      0.00 Bytes   0.00 bits/sec      sender
[ 5]   0.00-3.25    sec      729 MBytes  1.88 Gbits/sec      receiver
iperf Done.
```





## 2) 通过 UDP 测量带宽并获得抖动值。

```
$ iperf3 -c 172.16.6.130 -p 5000 -b 3G -t3 -u --get-server-output
Connecting to host 172.16.6.130, port 5000
[ 4] local 172.16.6.1 port 49809 connected to 172.16.6.130 port 5000
[ ID] Interval           Transfer     Bandwidth       Total Datagrams
[ 4]   0.00-1.00   sec    160 MBytes   1.34 Gbits/sec   115591
[ 4]   1.00-2.00   sec    163 MBytes   1.37 Gbits/sec   118045
[ 4]   2.00-3.00   sec    162 MBytes   1.36 Gbits/sec   117532
-----
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[ 4]   0.00-3.00   sec    485 MBytes   1.36 Gbits/sec   0.008 ms  3094/351159 (0.88%)
[ 4] Sent 351159 datagrams
Server output:
Server listening on 5000
Accepted connection from 172.16.6.1, port 54208
[ 5] local 172.16.6.130 port 5000 connected to 172.16.6.1 port 49809
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[ 5]   0.00-1.00   sec    145 MBytes   1.22 Gbits/sec   0.009 ms  1520/106563 (1.4%)
[ 5]   1.00-2.00   sec    149 MBytes   1.25 Gbits/sec   0.012 ms  776/108579 (0.71%)
[ 5]   2.00-3.00   sec    149 MBytes   1.25 Gbits/sec   0.024 ms  798/108657 (0.73%)
[ 5]   3.00-3.25   sec    37.8 MBytes   1.27 Gbits/sec   0.008 ms   0/27360 (0%)
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[ 5]   0.00-3.25   sec     0.00 Bytes   0.00 bits/sec   0.008 ms  3094/351159 (0.88%)
```

说明如下。

- ❑ -c: 客户端;
- ❑ 172.16.6.130: 服务端 IP 地址;
- ❑ -p: 服务端端口;
- ❑ -b: 最大发送数据的带宽 (如果无法发送, 会丢包);
- ❑ -t: 测试时间, 默认为 10s;
- ❑ --get-server-output: 在客户端获取服务端的统计数据。

在 mininet (<http://mininet.org>, 常作为 SDN 测试与开发的平台) 中会带有 iPerf 这个工具, 在软件网络中经常会用这个软件来测试网络性能和可达性。更多参数可以参考 <http://software.es.net/iperf/>。

#### 4.2.4 Fping

Fping (<http://www.fping.org>) 在 1992 年就被开发出来, 后来有很长一段时间停止了维护, 2011 年后由 David Schweikert 负责维护, 现在已经发布了 4.0 版本。Fping 也是一个开源软件。这个软件完全基于 ICMP 协议, 因此我们可以认为它是 ping 的一个增强版本。正是它的一些增强功能给我们带来了很多方便。比如, 网络工程师经常会对网络进行变更操作, 在这些变更中, 常常会涉及网络流量的变化。在流量的变化过程中, 如何才能快速地



跟踪和发现问题？通常的做法是持续地 ping 某些地址。Fping 提供了一个批量 ping 的方式，比如它可以从文件中读取一个需要 ping 的列表。

下面命令提供了一个持续 ping 示例，并且给出了一些常见的统计信息。

```
$ fping -l -e -Q1 www.taobao.com www.baidu.com
[10:34:43]
www.taobao.com : xmt/rcv/%loss = 1/1/0%, min/avg/max = 217/217/217
www.baidu.com : xmt/rcv/%loss = 1/1/0%, min/avg/max = 10.9/10.9/10.9
[10:34:44]
www.taobao.com : xmt/rcv/%loss = 1/1/0%, min/avg/max = 215/215/215
www.baidu.com : xmt/rcv/%loss = 1/1/0%, min/avg/max = 11.2/11.2/11.2
[10:34:45]
www.taobao.com : xmt/rcv/%loss = 1/1/0%, min/avg/max = 215/215/215
www.baidu.com : xmt/rcv/%loss = 1/1/0%, min/avg/max = 10.7/10.7/10.7
```

说明如下。

- -l: 持续发送报文，按 Ctrl+C 组合键可以终止；
- -e: 显示 RTT 值；
- -Q: 每间隔多少秒显示一个统计的结果。

Fping 使用 -g 参数后可以使用一个网段作为目的地址，Fping 会并行处理这些地址；还可以使用 -f 参数从文件中读取设备信息。更多的命令参数请参考 <http://www.fping.org/fping.1.html>。

Fping 可以结合 Smokeping (<http://oss.oetiker.ch/smokeping>) 提供基于 Web 的图形化 ping 测试结果，如图 4-3 所示。

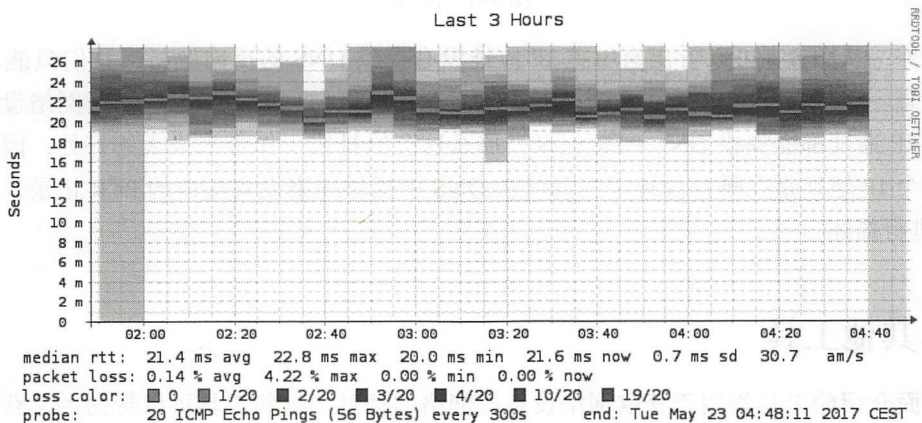


图 4-3 Smokeping (来自 <http://oss.oetiker.ch/smokeping/> 的 demo)

## 4.3 MTR

MTR (<http://www.bitwizzard.nl/mtr>) 是一个结合了 traceroute 和 ping 的网络诊断工具。



它是使用 GNU 发布的一个开源项目，能提供命令行界面和图形化界面的结果展示，目前的最新版本是 2016 年发布的 0.87。MTR 会先用 traceroute 的工作方式找到转发路径上的节点，然后使用 ICMP 持续地监控这些节点的情况，如图 4-4 所示。MTR 的输出除了图 4-4 的形式，还支持 JSON 或者是 XML 格式输出，这为后续处理这些数据提供了格式化后的数据结构。只需要在 mtr 命令后面添加 --json 或者是 --xml。关于其他命令行的参数，读者可以参考 <https://www.linode.com/docs/networking/diagnostics/diagnosing-network-issues-with-mtr>。

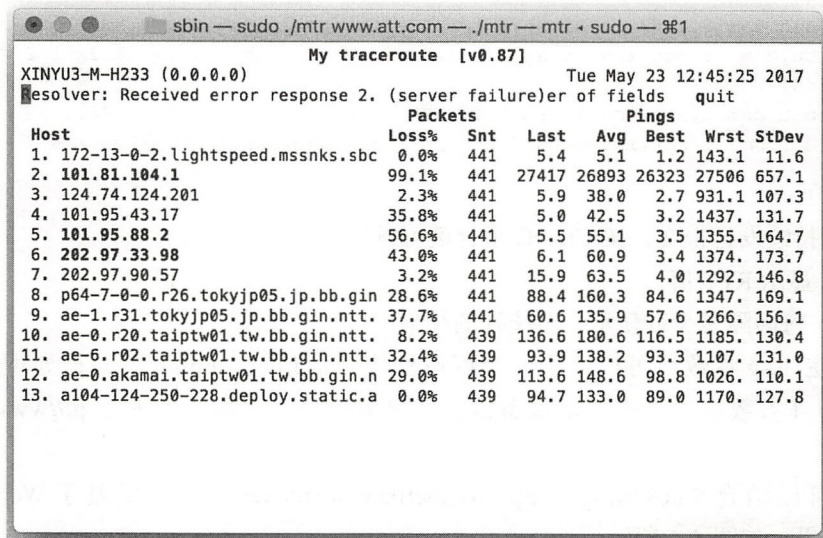


图 4-4 MTR

另外，从图 4-4 可以看出有一些 IP 地址没有提供 DNS 的反向解析，我们只能看到 IP 地址信息。如果在企业内部，这样的方式不太利于网络的诊断。如果我们对网络设备上的每一个 IP 地址都能提供 DNS PTR 记录，那么这会给排错工作带来极大的好处，因为不用记住哪个 IP 地址是在哪台设备上。如何在 DNS 中提供网络设备接口的 PTR 记录，这将在 6.3 节进行描述。

## 4.4 其他工具

前面介绍的工具均用于获取网络设备或网络的信息。这些工具可以帮助大家获得有效的数据。本节会简单介绍几个工具，这些工具要么是为了配合其他工具一起使用，要么是后续使用 REST API 做准备。

### 4.4.1 watch

watch 是一个非常小的工具，它不是以一个单独的软件包进行发布，而是放在了





procps (<https://gitlab.com/procps-ng/procps>, 软件 3.3.10 版本源代码存放位置) 这个软件包中进行发布。现在几乎所有的 Linux 发行版本都带有这个小工具。正如其名, watch 提供了在一个监视窗口下反复执行某一条命令的功能, 这个功能省去了反复输入命令的烦恼。虽然 watch 在网络设备上还不是很多, 但还是有一些网络设备上有了这个工具, 如 Arista EOS 系统。首先, 我们结合第 3 章的内容来看一个例子: 假设你想监视某台设备上的某条路由信息, 我们可以结合 SSH 远程执行命令的方式来完成。图 4-5 是下面命令运行的结果。

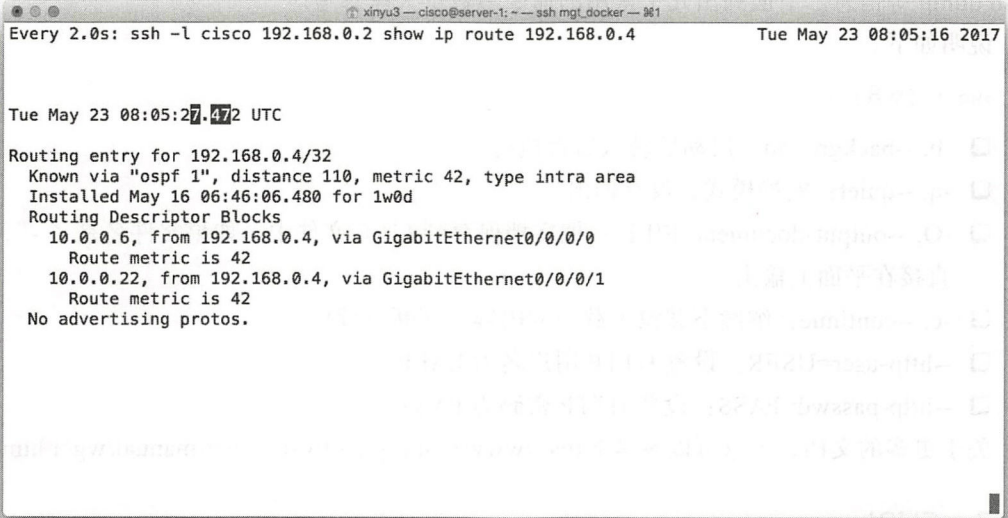
命令如下:

```
$ watch -d ssh -l cisco 192.168.0.2 show ip route 192.168.0.4
```

说明如下。

❑ -d: watch 的参数, 对每次不一样内容进行高亮显示。

-d 后都 SSH 命令了, 这些命令在第 3 章有说明, 这里不再赘述。登录设备需要输入密码, 如何才能实现使用 SSH 登录设备时不输入密码, 第 3 章介绍了两种方法。如果你处理的网络设备使用第 3 章提到的两种方式都不可行, 那么就需要通过一些脚本代码的方式来实现。这样的脚本代码如何实现, 读者在读完本书后就可以实现这样的功能。



```

xinyu3 - cisco@server-1: ~ - ssh mgt_docker - 81
Every 2.0s: ssh -l cisco 192.168.0.2 show ip route 192.168.0.4      Tue May 23 08:05:16 2017

Tue May 23 08:05:27.472 UTC
Routing entry for 192.168.0.4/32
  Known via "ospf 1", distance 110, metric 42, type intra area
  Installed May 16 06:46:06.480 for 1w0d
Routing Descriptor Blocks
  10.0.0.6, from 192.168.0.4, via GigabitEthernet0/0/0/0
    Route metric is 42
  10.0.0.22, from 192.168.0.4, via GigabitEthernet0/0/0/1
    Route metric is 42
No advertising protos.
  
```

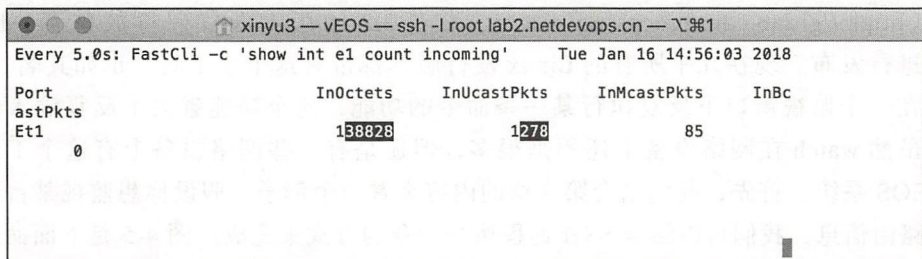
图 4-5 watch 结果

我们再来看看在 EOS 系统上使用 watch 命令。启动 EOS 的 Linux shell 后:

```
[admin@vEOS ~]$ watch -d -n 5 "FastCli -c 'show int e1 count incoming'"
```

这个命令可以每 5 秒来检查一下端口 ethernet1 入方向上的流量情况, 并对变化的数字进行高亮显示。其命令运行的结果见图 4-6。





Every 5.0s: FastCli -c 'show int e1 count incoming' Tue Jan 16 14:56:03 2018					
Port	astPkts	InOctets	InUcastPkts	InMcastPkts	InBc
Et1	0	158828	1278	85	0

图 4-6 Arista EOS watch 命令

## 4.4.2 Wget

Wget (<https://www.gnu.org/software/wget>) 是自由软件基金会 (Free Software Foundation) 资助的、基于 GNU 许可协议发布的软件，主要功能是基于 HTTP、HTTPS 和 FTP 协议下载文件，并且是一个非交互式的命令。因此，它很容易被放在脚本里执行。另外，Wget 可以跟踪网页 HTML 里面的链接进行下载，这也被称为递归下载。即使在链路质量很差的情况下，Wget 也会有很不错的表现，并且支持断点续传的功能，我们可以用它进行长时间的文件传输。如果网络设备支持 FTP 服务，我们就可以利用 Wget 从设备上下载日志或是 crash dump 等信息。下面列出几个常用的命令参数供大家参考。

说明如下。

\$wget [参数] [URL]

- ☐ -b, --background: 启动后转入后台执行。
- ☐ -q, --quiet: 安静模式，没有输出。
- ☐ -O, --output-document=FILE: 把文档保存到 FILE 文件中。如果文件名是“-”，则直接在平面上输出。
- ☐ -c, --continue: 继续下载没下载完的内容，即断点续传。
- ☐ --http-user=USER: 设置 HTTP 用户名为 USER。
- ☐ --http-passwd=PASS: 设置 HTTP 密码为 PASS。

关于更多的文档，大家可以参考 <https://www.gnu.org/software/wget/manual/wget.html>。

## 4.4.3 CURL

CURL 工具 (<https://curl.haxx.se>，目前版本为 7.54.0) 比 Wget 的功能更加丰富，它也是用于处理 URL 的工具。它支持文件的上传和下载，甚至可以认为是一个命令行界面的浏览器。它的参数非常丰富，远比 Wget 要多。由于 CURL 支持 HTTP/HTTPS 的 GET、POST、DELETE、PUT 这四个方法 (Wget 1.18 版本只支持 GET 方法和 POST 方法)，因此我们可以用 CURL 来实现功能相对简单的 RESTful API 调用，或者用它来作为一个 RESTful API 的调试工具，也还是很不错的。下面是通过 CRUL 命令从 Cisco FirePower 上



获取认证令牌的例子。更多关于这个工具的说明请参考 <https://curl.haxx.se/docs>。

```
$ curl -X POST -H "Authorization: Basic YWRtaW46QWRtaW4xMjM0NQ=="
https://10.74.82.25:443/api/fmc_platform/v1/auth/generatetoken -k -i

HTTP/1.1 204 No Content
Date: Tue, 23 May 2017 11:25:39 GMT
Server: Apache
Cache-Control: no-cache, no-store, must-revalidate, max-age=0
Accept-Ranges: bytes
Vary: Accept-Charset, Accept-Encoding, Accept-Language, Accept
X-auth-access-token: <略>
X-auth-refresh-token: <略>
USER_UUID: <略>
DOMAIN_ID: 111
DOMAIN_UUID: <略>
global: <略>
DOMAINS: [{"name": "Global", "uuid": "<略>"}]
Content-Length: 0
X-Frame-Options: SAMEORIGIN
```

## 4.5 小结

本章主要介绍了在 Linux 平台下的一些开源工具。由于篇幅的限制，有一些工具并没有详细地展开，但是笔者提供了参考链接。希望通过这些工具的介绍，读者能够了解到开源领域的一些工具，这些工具可以帮助大家不用编写代码就可以提高平时的工作效率。也正如在第3章提到的，NetDevOps 最常用的工作环境是 Linux 平台，因此熟悉更多的 Linux 工具也是很有帮助的。

在第5章，我们还会介绍一些 Linux 平台下的文本处理工具。结合本章介绍的一些工具的输出，我们就可以处理网络维护工作中遇到的很多问题。





## 处理网络设备输出的文本

对于网络设备而言，CLI（Command Line Interface，命令行界面）是网络设备最常用的人机交互界面。大家从开始学习网络到日常的工作几乎离不开这个界面。它是传统网络工程师最为熟悉的界面和工作环境。

在实际工作中，我们常常需要处理大量的 CLI 输出结果。也许我们需要统计一批设备的模块信息或者是接口卡的序列号，又或者是统计设备的路由信息等。对于这些信息的处理，往往不是只统计一两台设备，而是经常需要统计几十台甚至是上百台设备的 CLI 输出结果。对于这些内容的处理，有时候还需要间隔一段时间后，重复进行统计和分析。这些工作常常占用了我们很多工作时间。如何才能提高我们的工作效率？如何才能提高我们处理数据的准确性？我们是找专门的开发人员来做一个系统吗？这些需求常常是零散的，且又是希望立刻得到结果的。

因此，本章会介绍如何利用 Linux（也会涉及 Mac OS X）环境中一些常用且非常高效的文本处理工具。通过本章的介绍，读者可以使用这些工具进行一些简单的文本内容的获取、替换以及统计工作。相信这些工具可以大大地提高大家的工作效率。

### 5.1 正则表达式基础

在介绍这些工具之前，我们必须先要熟悉一下什么是正则表达式（Regular Expression）。正则表达式是后续提及的那些工具的基础。如果大家对这个部分的内容完全不了解，可以先跳过这个部分的内容，先从工具的使用开始。本章介绍的工具几乎都需要使用正则表达式。大家也可以在工具的使用过程中再回过头来查看这一节的内容。结合工具也许可以更好地掌握它。如果读者对正则表达式有所了解，那么希望通过这一节的介绍对大家做到查



漏补缺。

正则表达式（很多地方也简称为 regex）是处理文本的一种工具。正则表达式是 20 个世纪 50 年代左右在数学领域中最开始使用的一种工具。后来，随着计算机技术的兴起，在 UNIX 中诞生了 Perl 语言以及 grep 等工具。这些语言和工具就开始使用正则表达式来处理一些文本相关的工作。在很长一段时间内，正则表达式只出现在 UNIX 以及部分脚本语言中。但是，现在几乎每一种编程语言以及操作系统都支持正则表达式。除了编程语言，很多网络设备的配置或者 CLI 也支持正则表达式。例如，在网络设备上进行 BGP AS-PATH 过滤的时候，就可以使用正则表达式来表示 AS 号。对网络设备的命令行输出，也可以通过正则表达式来过滤结果，让网络工程师更直观地获取设备输出的信息。但是，不同的工具、语言或者平台对正则表达式的支持会有一些差异，这就需要大家在平时的实践中慢慢积累。不过，好在正则表达式的大部分内容还是相同的，而且在日常工作中，大部分只用到了正则表达式的基础功能。因此，本章涉及的正则表达式也尽量不会出现那些高级功能。如果读者需要了解那些高级功能，可以查阅一些相关的书籍或者文章。其中，网站 <http://regexr.com> 提供了一些不错的学习资料。

### 5.1.1 正则表达式到底是什么

在查找文本字符串的时候，大家经常会使用一些匹配的规则来获取信息，而正则表达式就是用来表达这些规则的描述语言。用正则表达式所描述的规则往往不太容易被人脑所解析，特别是对于初学者而言，那些复杂的正则表达式常常会让它们非常头疼。因为，它是一个面向机器更加友好的规则描述。一旦人们写出这些规则，通过一些工具或者编程语言应用到一大堆的文本上，它的处理效率是非常惊人的。

大家在使用 Windows 操作系统或者其他文本编辑工具的时候，应该用过“\*”或者是“?”来作为通配符进行模糊匹配。例如，如果你需要查找某个目录包含了哪些 Word 文件的时候，我们通常会使用 \*.doc 或者是 \*.docx 来表示它们。这里的“\*”就是一个通配符，在正则表达式中也会用到类似的通配符，只不过在正则表达式中，匹配方式更多而且更加灵活。

### 5.1.2 单字符的匹配

在字符的匹配上，字母的大小写是严格区分的，即“A”和“a”是完全不一样的字符。由于本章只涉及较为简单的正则表达式，因此，对于字符的匹配只涉及 ASCII 编码中的字符。虽然，现在很多语言和工具的正则表达式可以处理 UTF-8 或者其他编码格式，但是，网络设备的命令行输出结果的编码格式通常是 ASCII 编码。对于那些输出中文或者其他编码格式的设备，读者需要参考厂家的文档。

下面就从最简单的例子开始。在开始这个例子之前，我们给出一台 Cisco 路由器的接口信息，这里是全部的输出内容，在本章的后续章节还会用到这个内容。





```
Router# show ip interface brief
```

Interface	IP-Address	OK?	Method	Status	Protocol
GigabitEthernet0/1	unassigned	YES	unset	up	up
GigabitEthernet0/2	192.168.190.235	YES	unset	up	up
GigabitEthernet0/3	unassigned	YES	unset	up	up
GigabitEthernet0/4	192.168.191.2	YES	unset	up	up
TenGigabitEthernet2/1	unassigned	YES	unset	up	up
TenGigabitEthernet2/2	unassigned	YES	unset	up	up
TenGigabitEthernet2/3	unassigned	YES	unset	up	up
TenGigabitEthernet2/4	unassigned	YES	unset	down	down
GigabitEthernet3/1	unassigned	YES	unset	down	down
GigabitEthernet3/2	unassigned	YES	unset	down	down
GigabitEthernet3/3	unassigned	YES	unset	down	down
GigabitEthernet3/4	unassigned	YES	unset	down	down

### 1. 最简单的正则表达式

如果我们使用如下的正则表达式进行匹配：

```
GigabitEthernet0/2
```

毫无疑问，匹配结果应该是下面灰色的部分。

```
Router# show ip interface brief
```

Interface	IP-Address	OK?	Method	Status	Protocol
GigabitEthernet0/1	unassigned	YES	unset	up	up
GigabitEthernet0/2	192.168.190.235	YES	unset	up	up
GigabitEthernet0/3	unassigned	YES	unset	up	up
GigabitEthernet0/4	192.168.191.2	YES	unset	up	up
TenGigabitEthernet2/1	unassigned	YES	unset	up	up
TenGigabitEthernet2/2	unassigned	YES	unset	up	up
TenGigabitEthernet2/3	unassigned	YES	unset	up	up

<略>



**注意** 在本节中，为了能更好地说明问题，无论是否被正则表达式匹配都列出了所有的文本。匹配的文本以阴影（灰色）表示。

读者在进行正则表达式测试的时候，可以使用一些文本处理的应用程序。例如，sublime 是一个非常好用的且支持多平台（Windows、Mac OS X、Linux）的文本编辑工具。在 sublime 中，可以使用正则表达式进行文本的搜索和替换。下载地址为 <https://www.sublimetext.com>。当然，读者如果有其他的工具也是可以的。

也许你会觉得非常奇怪，这个难道是正则表达式吗？这分明就是一个简单的字符串而已。确实，这个正则表达式是最简单的形式，简单到和普通的文本搜索所使用的关键字没有任何区别。但是，这个也是我们最常用的一种形式。

### 2. 多匹配一些内容

如果我们要匹配一个（就是一个，而不是虚指某些）任意的字符，那么可以用“.”（点）





字符来表示。还是上面的文本，我们使用新的正则表达式如下：

```
GigabitEthernet.
```

这次使用的正则表达式为在 GigabitEthernet 后加了一个 “.” 字符。那么这次匹配到的文本如下：

```
Router# show ip interface brief
Interface          IP-Address      OK?    Method Status  Protocol
GigabitEthernet0/1 unassigned      YES    unset  up      up
GigabitEthernet0/2 192.168.190.235 YES    unset  up      up
GigabitEthernet0/3 unassigned      YES    unset  up      up
GigabitEthernet0/4 192.168.191.2   YES    unset  up      up
TenGigabitEthernet2/1 unassigned      YES    unset  up      up
TenGigabitEthernet2/2 unassigned      YES    unset  up      up
TenGigabitEthernet2/3 unassigned      YES    unset  up      up
<略>
```

我们可以看到，这次匹配的不仅有千兆（Gigabit）以太网接口，还有万兆（TenGigabit）以太网接口，并且还匹配到了紧跟着的数字。这里的 “.” 字符就是一个通配符，这个通配符可以匹配且只匹配一个任意的字符。

大家可以思考一下，在上面被省略掉的内容中是否还有匹配项呢？

### 3. 有选择地匹配一些内容

现在，我们再进行进一步修改之前的正则表达式，这次，我想匹配最后一位是 2 或者是 4 的千兆以太网接口，修改后的正则表达式如下：

```
^GigabitEthernet./[24]
```

匹配结果：

```
Router# show ip interface brief
Interface          IP-Address      OK?    Method Status  Protocol
GigabitEthernet0/1 unassigned      YES    unset  up      up
GigabitEthernet0/2 192.168.190.235 YES    unset  up      up
GigabitEthernet0/3 unassigned      YES    unset  up      up
GigabitEthernet0/4 192.168.191.2   YES    unset  up      up
TenGigabitEthernet2/1 unassigned      YES    unset  up      up
TenGigabitEthernet2/2 unassigned      YES    unset  up      up
TenGigabitEthernet2/3 unassigned      YES    unset  up      up
<略>
```

正则表达式最开始的字符 “^” 是一个元字符（元字符相关内容在后面会介绍）。在这里，它的含义是字符的起始。由于上面的文本还包含了 TenGigabitEthernet，如果不增加 “^” 这个元字符，那么将会匹配出更多的信息。这一点在本节的第二个例子中可以得到验证。

在正则表达式的编写中，要尽量保证匹配的结果是正好满足需求的。匹配条件过于宽松和过于严格的正则表达式都不是好的表达式。当然有时候为了使正则表达式更简单，适



当地放宽匹配条件也是可以的。

在正则表达式中, [24] 表示这里可以是 2, 也可以是 4。其中 “[” 和 “]” 都是元字符。这个表达式所要匹配的是一个字符的位置。这里比较容易误解的是把 24 看成一个数值, 而不是两个单独的字符。

大家再次思考一下, 在上面被略掉的内容是否还有匹配项呢?



正则表达式匹配的都是字符, 而不能识别数字的值。上面的例子中, 2 和 4 是字符 2 和字符 4, 而不是数值 2 和 4, 更加不是数值 24。这一点是需要大家特别注意。在正则表达式中, 并无法知道它的数值是多少 (但是, 某些工具是可以把这个字符类型变成数值类型的)。所有的内容在正则表达式中都是字符形式体现的。

#### 4. 非打印字符

在某些时候, 匹配的字符并不是可以直接打印出来的字符, 如换行符、回车符等。表 5-1 列出了一些非打印字符的表示方法。对于非打印字符, 为了能很好地表示它们, 可使用转义方式来表达。转义符号为 “\”。如果读者学习过 C 语言编程, 那么对这个是很容易理解的。

表 5-1 非打印字符

字 符	说 明
\f	用于匹配一个换页符
\n	用于匹配一个换行符
\r	用于匹配一个回车符
\s	用于匹配任何一个空白字符, 包括空格、制表符、换页符等, 等价于 [\f\n\r\t\v]
\S	用于匹配任何一个非空白字符, 等价于 [^\f\n\r\t\v]
\t	用于匹配一个制表符
\v	用于匹配一个垂直制表符



在网络设备的输出结果中, 有的厂家的换行符号为 \r\n, 而有的厂家的换行符为 \n。大家在实际情况中可能需要注意这一点。

在处理网络设备的输出时, \s 是非常常用的符号, 因为它既包含了空格, 又包含了制表符 (常常被称为 tab 键)。很多厂家的 CLI 输出是会夹杂着空格和制表符的。


#### 5. 特殊字符

特殊字符 (见表 5-2) 在正则表达式中有特殊的作用或者含义。如果要匹配这些字符, 也需要在其前面加上 “\” 作为转义使用。



表 5-2 特殊字符

特别字符	说 明
\$	用于匹配输入字符串的结尾位置
()	用于一个子表达式的开始与结束的位置
*	用于匹配前面的子表达式零次或多次
+	用于匹配前面的子表达式一次或多次
.	用于匹配除换行符 \n 之外的任何单字符
[]	用于标记一个中括号表达式的开始与结束
?	用于匹配前面的子表达式零次或一次，或指明一个非贪婪限定符
\	将下一个字符标记为特殊字符、原义字符、向后引用、八进制转义符
^	用于匹配输入字符串的开始位置；或者在方括号表达式中使用，此时它表示该字符集合的非集合
{	用于标记限定符表达式的开始
	用于指明两项之间的一个选择，即“或”的含义

 **提示** ☐ \b 为单词的定位符，匹配一个单词的边界，即字与空格间的位置；  
☐ \B 正好相反，表示这个位置前后一定不是空格，是非单词的边界。  
在本节的第三个例子中，正则表达式也可以写成“\b GigabitEthernet./[24]”。使用 \b 这个定位符也是比较常用的一种方式了。

5.1.3 多字符的匹配与次数匹配

在很多情况下，需要匹配多个字符的形式，这个时候就需要用到匹配多个连续重复出现的字符或者是字符集合。为了说明这个问题，我们拿网络工程师最常用的一种场景——获取文本中的 IP 地址作为例子。需要被匹配的文本还是 5.1.2 节中的文本，即一台路由器的接口信息。上面的文本中只有 2 个 IP 地址。下面来看看我们是否可以通过一个正则表达式获取到 IP 地址。

1. 第一次匹配 IP 地址

使用正则表达式：

```
[0-9][0-9][0-9].[0-9][0-9][0-9].[0-9][0-9][0-9].[0-9][0-9][0-9]
```

注意，这个表达式中是没有空格的。

匹配结果：

```
Router# show ip interface brief
```

Interface	IP-Address	OK?	Method	Status	Protocol
GigabitEthernet0/1	unassigned	YES	unset	up	up
GigabitEthernet0/2	192.168.190.235	YES	unset	up	up
GigabitEthernet0/3	unassigned	YES	unset	up	up





```
GigabitEthernet0/4 192.168.191.2 YES unset up up
TenGigabitEthernet2/1 unassigned YES unset up up
TenGigabitEthernet2/2 unassigned YES unset up up
TenGigabitEthernet2/3 unassigned YES unset up up
...略
```

我们发现，上面的正则表达式只匹配了一个 IP 地址，另外一个 IP 地址并没有匹配到，且使用的表达式非常长。

为什么我们只能匹配第一个 IP 地址，而不能匹配第二个呢？那是因为，正则表达式限制了 IP 地址必须是一个三位四段的 IP 地址，而第二个 IP 地址的最后一位只有一个字符 2，因此，无法匹配到。

## 2. 修改一下匹配 IP 地址表达式的 bug

我们改写表达式为

```
[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+
```

匹配结果：

```
Router# show ip interface brief
Interface      IP-Address      OK?  Method Status  Protocol
GigabitEthernet0/1 unassigned      YES  unset  up      up
GigabitEthernet0/2 192.168.190.235 YES  unset  up      up
GigabitEthernet0/3 unassigned      YES  unset  up      up
GigabitEthernet0/4 192.168.191.2  YES  unset  up      up
TenGigabitEthernet2/1 unassigned      YES  unset  up      up
TenGigabitEthernet2/2 unassigned      YES  unset  up      up
TenGigabitEthernet2/3 unassigned      YES  unset  up      up
...略
```

每个字符匹配的后面使用了“+”，表示前面的这个数字可以重复 1 次到无穷次，这样就可以匹配到 192.168.191.2 这个字符串了。

除了“+”以外还有其他表示重复的表达式，如表 5-3 所示。

表 5-3 匹配次数限定符

限定符	说 明
{n}	表达式固定重复 $n$ 次，比如：“[0-9]{2}”相当于 “[0-9][0-9]”
{m, n}	表达式尽可能重复 $n$ 次，至少重复 $m$ 次：“F1{1,3}”可以匹配“F1”或“F11”或“F111”
{m, }	表达式尽可能地多匹配，至少重复 $m$ 次：“Ethernet\[0-9]{2,}”可以匹配“Ethernet12”，“Ethernet1234”……
?	表达式尽可能匹配 1 次，也可以不匹配，相当于 {0, 1}
+	表达式尽可能地多匹配，至少匹配 1 次，相当于 {1, }
*	表达式尽可能地多匹配，最少可以不匹配，相当于 {0, }



### 3. 再次完善匹配 IP 地址的表达式

基于表 5-3，我们对 IP 地址的正则表达式又可以优化为

```
[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
```

这个表达式可以很好地匹配出上面文本中的 IP 地址。不过，你也许已经意识到了，这个表达式其实包含了更多的内容。例如，300.300.300.300 也是可以被匹配到的。但是在网络设备的输出中几乎不会出现上述的非法 IP 地址形式，那么这个形式的正则表达式是可以满足大部分的需求的。如果你有兴趣，可以进一步优化这个表达式，使其可以更加精确地匹配到一个 IP 地址。在这里先不马上给出一个答案，你可以在后续的内容中找到更加优化的 IP 地址正则表达式，或者也可以通过互联网查询或者寻求到其他人对 IP 地址表达式的优化。

#### 5.1.4 在网络设备上的正则表达式

正则表达式是一个很好的处理文本信息的工具，已经被广泛地应用到了很多平台上，甚至是很多的网络设备中。例如，下面是一台 Cisco 路由器的配置中关于 BGP AS-PATH 过滤的一个例子：

```
ip as-path access-list 1 deny ^123.*
router bgp 109
network 172.18.0.0
neighbor 172.19.6.6 remote-as 123
neighbor 172.23.1.1 remote-as 47
neighbor 10.125.1.1 filter-list 1 out
```

又如下面是一台 Juniper 的路由器对 CLI 的输出结果直接使用正则表达式进行匹配的结果。在 JUNOS 平台上，match 命令和 grep 命令在很多场景下是一致的，甚至可以把 match 命令替换为 grep 命令。

```
user@host> show configuration | match "at-[25]"
at-2/1/0 {
at-2/1/1 {
at-2/2/0 {
at-5/2/0 {
at-5/3/0 {
```

随着网络设备的控制平面越来越多地转向基于 Linux 的操作系统，在命令行中也更容易使用正则表达式来处理设备输出的结果。

熟悉和掌握正则表达式是提高效率的一个很好的途径。正则表达式的内容不止本节包含的这些内容，还有一些高级的应用（本节并没有涉及），但是，本节介绍的内容可以涵盖平时工作中大部分的应用。本书的后续内容将会涉及更多的关于正则表达式的内容。



## 5.2 使用 grep 进行搜索与获取信息

### 5.2.1 什么是 grep

grep (Globally search a Regular Expression and Print, 以正则表达式进行全局搜索和打印) 是一个 UNIX 下基于命令行的常用工具。它可以基于一个或者多个正则表达式对文本内容进行搜索, 并输出经过匹配处理后的信息。常用的 grep 工具主要有两个版本: 一个是 GNU grep (<https://www.gnu.org/software/grep>), 几乎所有的 Linux 发行版使用这个版本; 另一个是 BSD grep (<https://wiki.freebsd.org/BSDgrep>), FreeBSD、OpenBSD 以及 MAC OSX 均使用这个版本。这两个版本只存在非常小的差别。

grep 家族还包括很多扩展的工具集, 如 fgrep、egrep、zgrep。扩容工具 fgrep 比 egrep、grep 能提供更高的搜索效率, 但是它只能提供相对简单的正则表达式。工具 egrep 能比 grep 提供更加丰富的正则表达式的匹配模式和方法。扩展工具 zgrep 可以在 zip 文件中直接提供搜索功能, 使之不需要解压 zip 文件就可以进行搜索。

Linux 与 Mac OS X 发行版本默认都集成了 grep 工具, 图 5-1 是 Mac OS X 中集成的 grep 版本信息。当前版本的 grep 工具可以通过 -E、-F、-Z 分别使用 egrep、fgrep、zgrep 工具, 使用命令方式也是可以的, 只不过这已经不是推荐的方式了。

另外, grep 不仅仅可以作为工具来使用, 还可以被放在 shell 脚本中执行。其搜索匹配的结果还返回了状态码: 如果查找成功会返回 0, 反之如果无法获得搜索结果的值则会返回 1。

grep 是一个基于“行”的搜索工具, 以“行”为单位进行操作。行与行之间的关联性并不是 grep 善于处理的内容。但是, 某些情况下可以通过多次使用 grep 工具来达到目的, 本节会有一些具体例子。

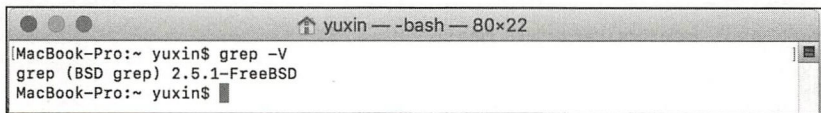


图 5-1 Mac OS X 下的 grep 版本信息

### 5.2.2 命令选项的解释

其基本语法如下:

```
grep [选项] [正则表达式] [文件]
```

grep 的选项 (大部分选项包含短选项和长选项, 其功能完全一样) 会比较多, 为了便于读者快速了解, 表 5-4~表 5-6 提供了它们的分类和解释。



表 5-4 grep 选项的匹配控制

选项 (参数)	解 释
-e 正则表达式 --regexp= 正则表达式	在选项中可以出现多个“-e”的内容，每个正则表达式之间是或的关系。如果表达式中出现空格，需要使用双引号来包括表达式
-i --ignore-case	忽略大小写
-f 文件名 --file= 文件名	从文件中读取正则表达式，每一行为一个表达式。如果文件内容是空的，那么什么也不匹配
-v --invert-match	对正则表达式匹配的内容取反操作，即选择那些没有匹配到的行
-E 正则表达式 --extended-regexp= 正则表达式	使用扩展的正则表达式。和使用 egrep 一致

表 5-5 grep 选项的输出结果控制

选项 (参数)	解 释
-n --line-number	在输出结果中显示原始文件匹配到的行号
-c --count	计算匹配的行的数量，即使一行中有两次以上的匹配也只会算一次。注意是行的数量，而不是匹配的数量
-l --files-with-matches	只显示匹配的文件名。在多文件查找的时候会有意义
-L --files-without-match	只显示没有匹配的文件名。和上一个选项正好相反
-m 行数 --max-count= 行数	指定最多输出多少个匹配行。如果一行中有多个匹配项，也只算一次，因为需要的是行的数量
-o --only-matching	只输出匹配的字符串内容
-H --with-filename	在每一行输出前添加文件名。在多文件操作时会默认添加这个参数
-h --no-filename	在每一行输出前不添加文件名。在单文件操作时，默认添加这个参数
-A 行数 --after-context= 行数	输出匹配行及其后的行内容。如果行数的值为 0，则输出匹配行。 注意：参数后可以紧跟数字，也可以在参数后加空格再加数字。例如，-A1 和 -A 1 都是可以的。在使用参数 -A、-B、-C 时候，默认会添加“--”作为分割行信息。 可以使用 --group-separator= 字符串来修改这个分割行信息（只有 GNU grep 版本支持，Mac OS X 下的 grep 不支持） 还可以使用 --no-group-separator 取消分割行信息（只有 GNU grep 版本支持，Mac OS X 下的 grep 不支持）
-B 行数 --before-context= 行数	和 -A 相反。提供匹配前的内容输出
-C 行数 --context= 行数	提供匹配前和匹配后的行内容。参数可以简化为“- 行数”，如“-2”

表 5-6 grep 选项的输入控制

选项 (参数)	解 释
-r --recursive	如果是目录, 读取目录中的所有文件; 如果存在子目录, 继续递归处理。这在一个目录下搜索内容很有用
-I --binary-file=without-match	忽略二进制文件

在接下来的内容中, 我们会通过例子来解释上面的大部分选项。下面的例子将不再对使用的选项进行解释, 读者可查询表 5-4~表 5-6。

### 5.2.3 匹配控制

5.1.2 节有一段 Cisco IOS 设备接口信息的输出结果, 这段 CLI 的输出保存在 [https://github.com/netdevops-engineer/newbie\\_book/tree/master/Chapter05](https://github.com/netdevops-engineer/newbie_book/tree/master/Chapter05), 文件名为 if\_info\_ios.txt。我们将使用这个文件作为例子。

#### 1. 查找有 IP 地址行

```
$ grep -E
"((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)"
if_info_ios.txt
GigabitEthernet0/2    192.168.190.235 YES    unset up    up
GigabitEthernet0/4    192.168.191.2   YES    unset up    up
```

这里给出了关于匹配 IP 地址的正则表达式, 这个表达式比 5.1.3 节中的要更加精准。也许你会觉得这么长的正则表达式不容易记住, 其实你可以把这个 IP 地址的正则表达式保存到系统变量里, 每次可以通过系统变量来使用。在 Linux 的 Bash 中, 可以把这个变量放在 \$HOME/.bash\_profile 里, 这样每次登录系统的时候都会加载。在本章的后续内容中, 如果使用到 IP 地址的正则表达式都会用变量 \$IP\_RE 来表示。

```
$ export
IP_RE="((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)"
$ grep -E $IP_RE if_info_ios.txt
GigabitEthernet0/2    192.168.190.235 YES    unset up    up
GigabitEthernet0/4    192.168.191.2   YES    unset up    up
```

如果你不愿意用这么复杂的正则表达式, 在这个例子中可以使用如下命令:

```
$ grep -v unassigned if_info_ios.txt
Router# show ip interface brief
Interface            IP-Address      OK?    Method Status      Protocol
GigabitEthernet0/2    192.168.190.235 YES    unset up        up
GigabitEthernet0/4    192.168.191.2   YES    unset up        up
```

#### 2. 查找千兆接口的行

使用参数 “i” 可以忽略文本中的大小写。

```
$ grep -i "\bgigabit" if_info_ios.txt
GigabitEthernet0/1      unassigned      YES      unset  up        up
GigabitEthernet0/2      192.168.190.235 YES      unset  up        up
GigabitEthernet0/3      unassigned      YES      unset  up        up
GigabitEthernet0/4      192.168.191.2  YES      unset  up        up
GigabitEthernet3/1      unassigned      YES      unset  down      down
GigabitEthernet3/2      unassigned      YES      unset  down      down
GigabitEthernet3/3      unassigned      YES      unset  down      down
GigabitEthernet3/4      unassigned      YES      unset  down      down
```

### 5.2.4 输出结果控制

1) 统计有 IP 地址接口的数量。使用参数 “-c” 来统计行的数量。

```
$ grep -c -E $IP_RE if_info_ios.txt
2
```

2) 给出有 IP 地址的接口名称。

```
$ grep -E $IP_RE if_info_ios.txt | grep -o -E "\w+[0-9]\{0-9\}"
GigabitEthernet0/2
GigabitEthernet0/4
```

这里使用了两次 `grep`。使用参数 “-o” 最后只输出接口的名称，而去掉了行中的其他信息。

3) 在配置中查找哪些接口配置了 ISIS 协议。Cisco IOS 配置文件（文件名为 `router_isis_ios.txt`）：

```
router isis
  net 49.0001.0000.0000.0001.00

interface ethernet0/0
  ip router isis
  ip address 172.17.1.1 255.255.255.0

interface ethernet0/1
  ip address 172.16.1.1 255.255.255.0

interface serial2/0
  ip router isis
  ip address 192.168.1.1 255.255.255.0

interface serial5/0
  ip address 172.21.1.1 255.255.255.0
```

上面是 Cisco IOS 关于 ISIS 的简化配置，用于作为命令的测试。在 Cisco IOS 配置文件中，`ip router isis` 是包含在接口（interface）中的。

```
$ grep -e"interface" -e"ip router isis" router_isis_ios.txt | grep -B1 "ip router
  isis" | grep -o -E "\w+[0-9]\{0-9\}"
```



## 82 ❖ 第二篇 基础篇

```

ethernet0/0
serial2/0

```

这里使用了三次 `grep` 命令。为了便于理解，下面给出前两次的 `grep` 的结果。

```

$ grep -e"interface" -e"ip router isis" router_isis_ios.txt
interface ethernet0/0
    ip router isis
interface ethernet0/1
interface serial2/0
    ip router isis
interface serial5/0

```

我们可以看到，第一次使用 `grep` 命令后只剩下了接口和 ISIS 相关的行内容。

```

$ grep -e"interface" -e"ip router isis" router_isis_ios.txt | grep -B1 "ip router
    isis"
interface ethernet0/0
    ip router isis
--
interface serial2/0
    ip router isis

```

我们可以看到，这时只剩下了接口名称和 “ip router isis” 这两行配置了，然后在这个基础上再取出接口名。

刚才我们是从 Cisco IOS 的配置中获取那些运行了 ISIS 协议的接口信息。这里我们再给一个从 Juniper JUNOS 配置文件（文件名为 `router_isis_junos.txt`）中获取 ISIS 接口信息的例子，这里我们需要用到 `set` 命令的显示方式。

```

set interfaces ge-1/2/0 unit 0 description to-R1
set interfaces ge-1/2/0 unit 0 family inet address 10.0.0.2/30
set interfaces ge-1/2/0 unit 0 family iso
set interfaces ge-1/2/1 unit 0 description to-R2
set interfaces ge-1/2/1 unit 0 family inet address 10.0.1.2/30
set interfaces ge-1/2/2 unit 0 description to-R3
set interfaces ge-1/2/2 unit 0 family inet address 10.0.2.2/30
set interfaces lo0 unit 0 family inet address 192.168.0.2/32
set interfaces lo0 unit 0 family iso address 49.0001.0192.0168.0002.00
set protocols isis interface ge-1/2/0.0
set protocols isis interface ge-1/2/2.0
set protocols isis interface lo0.0

```

熟悉 JUNOS 配置的读者也许发现了上面的配置是存在一些问题的。我们如何能快速地找出上面的问题呢？

命令 1:

```

$ grep -e "iso" router_isis_junos.txt | grep -o -E "interfaces \S+"
interfaces ge-1/2/0
interfaces lo0

```

命令 2:

```
$ grep -e "protocols isis" router_isis_junos.txt | grep -o -E "interface \S+"
interface ge-1/2/0.0
interface ge-1/2/2.0
interface lo0.0
```

通过这两个命令，就可以很容易发现：ge-1/2/2 接口并没有配置 family iso，而在 protocols isis 中却配置了接口 ge-1/2/2，那么这个接口是无法正常运行 ISIS 协议的。

当然，如果这里存在很多的接口，也很难一眼就看出问题来。不过，我们这里只使用了 grep 一个工具。随着介绍的工具越来越丰富，我们将可以更加快速地找到问题。

## 5.2.5 输入控制

目前我们已经使用了三个文件，它们分别是 if\_info\_ios.txt、router\_isis\_ios.txt、router\_isis\_junos.txt。现在我们希望在这三个文件中查询它们包含了哪些 IP 地址。

```
$ grep -o -E $IP_RE -r .
./if_info_ios.txt:192.168.190.235
./if_info_ios.txt:192.168.191.2
./router_isis_ios.txt:172.17.1.1
255.255.255.0
./router_isis_ios.txt:172.16.1.1
255.255.255.0
./router_isis_ios.txt:192.168.1.1
255.255.255.0
./router_isis_ios.txt:172.21.1.1
255.255.255.0
./router_isis_junos.txt:10.0.0.2
./router_isis_junos.txt:10.0.1.2
./router_isis_junos.txt:10.0.2.2
./router_isis_junos.txt:192.168.0.2
```

显然，这个结果不是非常令人满意，因为 255.255.255.0 这个子网掩码也包含在其中了。但是，我们至少用了一个命令就列出了多个文件中的 IP 地址，并且给出了是由哪个文件包含的。如果要把 255.255.255.0 这个子网掩码剔除掉，也许需要修改前面的 IP 正则表达式。当然，我们在这里可以在用一个 grep -v 来删除它。命令如下：

```
$ grep -o -E $IP_RE -r . | grep -v 255.255.255.0
./if_info_ios.txt:192.168.190.235
./if_info_ios.txt:192.168.191.2
./router_isis_ios.txt:172.17.1.1
./router_isis_ios.txt:172.16.1.1
./router_isis_ios.txt:192.168.1.1
./router_isis_ios.txt:172.21.1.1
./router_isis_junos.txt:10.0.0.2
./router_isis_junos.txt:10.0.1.2
./router_isis_junos.txt:10.0.2.2
./router_isis_junos.txt:192.168.0.2
```

但是，在处理的这些文件中，子网掩码是不是不止 255.255.255.0 这一个呢？读者可以思考一下这个问题。如果一时找不到答案也没有关系，毕竟我们不是用 `grep` 来处理所有的问题，也许用其他工具处理起来会更加方便。

`grep` 的主要功能是进行基于行的搜索功能，而且其运行效率非常高。如果读者还希望了解更多关于 `grep` 的应用请参考 <https://www.gnu.org/software/grep/manual/grep.html>。

## 5.3 使用 `awk` 进行文本处理

5.2 节介绍的 `grep` 是一个行搜索工具，而本节要介绍的 `awk` 则是一个基于列的文本处理工具，但是它读取文本的时候又是按照行进行读取的。每读完一行数据，`awk` 都会把这行数据处理成若干列，并保存到列字段中，最后按照字段（也就是列）输出数据。`awk` 是一个非常优秀的文本处理工具，在处理类似表格的文本时会更加得心应手。网络设备的 CLI 命令输出，很多时候是采用空格或制表符来分割文本的。对于这样的内容，用 `awk` 处理会非常方便且高效。

### 5.3.1 认识一下 `awk`

最早版本的 `awk` 可以追溯到 1977 年，它由 Alfred Aho（阿尔弗雷德·艾侯）、Peter Weinberger（彼得·温伯格）和 Brian Kernighan（布莱恩·柯林汉）三个人共同完成。这个软件的名字来自于这三个人姓的首字母。`awk` 有极其强大的功能，甚至具备了一个完整语言所能拥有的功能，实际上 `awk` 确实有自己的语言体系。现在我们常用的 `awk` 是 GNU 版本，有时候也称它为 `gawk` (<https://www.gnu.org/software/gawk>)。设计 `awk` 的目的是进行文本处理，它在很多方面和 Linux shell 非常类似。



**提示** `awk` 是一种处理文本文件的语言。它将文件作为记录序列处理。一般情况下，文件内容的每行都是一个记录。每行内容都会被分割成一系列的域，因此，我们可以认为一行的第一个词为第一个域，第二个词为第二个，依此类推。`awk` 程序是由一些处理特定模式的语句块构成的。`awk` 一次可以读取一个输入行。对于每个输入行，`awk` 解释器会判断它是否符合程序中出现的各个模式，并执行符合的模式所对应的动作。——Alfred Aho《AWK 编程》

### 5.3.2 `awk` 执行方式与语法

`awk` 通常可以采用两种方式来执行：一种方式是通过命令行的参数来运行；另一种方式则是把执行的内容写在一个文本中，然后通过 `awk` 来执行这段 `awk` 代码。本书不涉及第二种方式，在第一种方式中，其参数也是由一段小小的代码构成的。



下面是这两种方式的命令行语法规则。

```
$ awk '匹配条件 <动作>' 文件
```

**解释：**匹配条件的内容主要采用正则表达式来实现。动作主要是一些输出方式。最后会加上一个或多个文件。

```
$ awk -f <代码文件> 文件
```

**解释：**这里的代码文件其实就是第一种方式引号内的内容，对于那些复杂的操作，我们就会写成这样的形式。

awk 在处理文本内容的时候通常会遇到如下三种情况。

第一种是读取一行内容就处理一行内容，当读取完成后处理也完成了。其通常用于对文本内容的重新格式化操作，或可以认为是从文本中截取一部分需要的信息。

第二种是先读取文本内容，待其全部读取完成后对文本内容进行二次加工的操作。最常见的是对内容进行一些统计分析，比如对某列进行计数或者累加等计算。

第三种是结合了第一种方式和第二种方式。因此，熟悉了第一和第二种处理方式，第三种也成了顺理成章的事了。接下来，我们就通过几个例子来处理一些设备 CLI 输出的内容。

这里我们将用 awk 再次处理一下在 5.2 节 grep 处理过的几个文本，大家可以思考一下这两个工具的区别。

### 5.3.3 截取部分信息

这里我们还使用 5.2.3 节中 if\_info\_ios.txt 的内容，并且用 awk 来重复实现 grep 的功能。

查找有 IP 地址的行，并且只输出接口和 IP 地址的信息：

```
$ awk '$2 ~ /[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+/ {print $1,$2}' if_info_ios.txt
GigabitEthernet0/2 192.168.190.235
GigabitEthernet0/4 192.168.191.2
```

默认 awk 会使用空格进行列的分割，\$0 代表整行，而 \$1、\$2 等分别代表第一列、第二列等。匹配规则也使用了正则表达式，两个“/”内为正则表达式，“~”表示后使用正则表达式，“{ }”中是输出的内容。if\_info\_ios.txt 是文件名。

可以看到，awk 可以基于列进行查找，也可以进行列的输出，比 grep 要灵活很多。

当然，awk 的匹配规则也可以取反。其方法是在“~”前加一个“!”。具体如下：

```
$ awk '$2 !~/unassigned/ {print $1,$2}' if_info_ios.txt
Router# show
Interface IP-Address
GigabitEthernet0/2 192.168.190.235
GigabitEthernet0/4 192.168.191.2
```

大家应该发现了，输出结果中第一行的内容也许是我们不需要的，第二行的内容是字段名称的信息。如何跳过第一行以及第二行在 5.3.4 节中会提到。或者使用多条件匹配：

```
$ awk '$1 !~/Gi/ && $2 !~/unassigned/ {print $1,$2}' if_info_ios.txt
GigabitEthernet0/2 192.168.190.235
GigabitEthernet0/4 192.168.191.2
```

5.3.4 使用内置变量

awk 有很多内置变量，可以方便大家使用，其实 5.3.3 节中的 \$1 和 \$2 就是内置的变量。表 5-7 列出了一些常用的内置变量。

表 5-7 awk 部分内置变量

变量名 (注意 \$)	说 明
\$0	当前行所有的内容
\$1~\$n	每一列数据，默认使用空格分隔列。分隔符是一个空格，在文本中一个或多个空格，或 Tab 符都可以将数据分为一列
FS	指定字段分隔符，默认是空格
NF	当前行中字段的个数，也就是列数
NR	已经读取的记录数，就是行号，编号从 1 开始。如果是多个文件，则是累加计数
FILENAME	输入文件的名字
FNR	当前记录数，每个文件单独计数

在 5.3.3 节的例子中，如果需要剔除第一行和第二行的内容可以使用：

```
$ awk 'BEGIN{NR>2} /[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+/ {print $2}' if_info_ios.txt
192.168.190.235
192.168.191.2
```

BEGIN 是关键字，表示在读取记录前进行的操作。另外，关键字 END 表示全部扫描完成后进行的操作，END 相关内容会在 5.3.5 节用到。

5.3.5 对特定内容进行统计分析

我们在 5.2.4 节的例子中用 grep 统计了 IP 地址的数量。在 awk 中，我们先实现一个类似的例子。

```
$ awk -v counter=0 '{if ($2 ~/[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+/) {counter++}}
END{print counter}' if_info_ios.txt
2
```

-v 定义一个变量，并且确定初始值是 0。其后面是一个简单的逻辑过程，假如第二列可以匹配到 IP 地址，那么 counter 计数器就加 1，这里的 “++” 和 C 语言是一样的。END 后的内容表示把这个 counter 的值打印出来。END 的意思是，等前面对文件的扫描全部结

束后再进行的操作。

从这个例子看，awk 好像比 grep 要麻烦很多。但是，如果我们需要一次统计出有 IP 的接口数量及没有 IP 的接口数量，并且还要打印出所有的接口与 IP 地址的对应关系，以及所有接口在文本中所对应的行号（这个需求确实有点长），可以使用如下命令：

```
$ awk -v ip_counter=0 \
    -v noip_counter=0 \
    '{if ($2 ~/[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)/} \
    {ip_counter++; print NR, $1, $2} \
    else if($2 ~/unassigned/) \
    {noip_counter++;}} \
    END{print "ip inf counter: ",ip_counter; \
    print "no ip inf counter:", noip_counter }' \
    if_info_ios.txt
4 GigabitEthernet0/2 192.168.190.235
6 GigabitEthernet0/4 192.168.191.2
ip inf counter: 2
no ip inf counter: 10
```

上面的例子有一点复杂，不过它的逻辑还是很清晰的。这个例子的命令也会放在 GitHub 中，文件名是 awk\_command.md。

这里我们再看一个例子。如果我们需要统计 if\_info\_ios.txt 这个文件中有多少个接口的状态为 down，有多少个接口的状态为 up，可以使用如下命令：

```
$ awk '/Gi/ {inf[$5]++} END{for (x in inf) {print x, inf[x]}}' if_info_ios.txt
down 5
up 7
```

或者

```
$ awk '/Gi/ { inf[$5]++} END{print "down", inf["down"]; print "up", inf["up"]}' \
    if_info_ios.txt
down 5
up 7
```

在这个例子中，我们用到了 awk 的数组变量。首先，我们使用 “/ Gi /” 对接口进行了匹配操作；其次，使用 inf[\$5]++，对数组名为 inf、下标为 \$5 中的内容进行计数（这里将会是 “up” 或者是 “down”，计数方式在本节开始用到了）；最后把结果打印出来。在打印的时候用了两种方法：一种是不知道下标名；另一种是明确知道下标名。

下面我们再看一个例子。假设我们想知道某台设备所有接口的入流量和出流量的和。通过统计一台设备的出入流量的情况，可以初步判断出这台设备有没有丢包。通过这个办法可以来筛查哪些设备可能出现了问题。（如果设备上配置了一些 ACL，那么这种方法也许会不太准确。）首先，我们已经从设备上获得了所有接口的流量信息<sup>①</sup>。

<sup>①</sup> 在 GitHub 的 Chapter05 目录下有一个 inf\_traffic\_ios.txt 文件，这个文件记录了所有接口的流量信息。



这里通过 `grep` 命令过滤出接口的出入流量情况。

```
$ grep -e "packets input" -e "packets output" -e "line protocol" inf_traffic_ios.txt
GigabitEthernet0/0 is up, line protocol is up
    404923 packets input, 29499523 bytes, 0 no buffer
    369489 packets output, 34928106 bytes, 0 underruns
GigabitEthernet0/1 is administratively down, line protocol is down
    11710 packets input, 1040625 bytes, 0 no buffer
    19787 packets output, 1508273 bytes, 0 underruns
GigabitEthernet0/2 is up, line protocol is up
    148477 packets input, 14084361 bytes, 0 no buffer
    252803 packets output, 23131470 bytes, 0 underruns
GigabitEthernet0/3 is up, line protocol is up
    164504 packets input, 14518699 bytes, 0 no buffer
    213198 packets output, 20325631 bytes, 0 underruns
Loopback0 is up, line protocol is up
    0 packets input, 0 bytes, 0 no buffer
    2 packets output, 381 bytes, 0 underruns
Loopback1 is up, line protocol is up
    0 packets input, 0 bytes, 0 no buffer
    1 packets output, 28 bytes, 0 underruns
```

现在，我们将使用 `awk` 命令来对每个接口的出入流量进行求和的操作。

```
$ awk '/packets / {sum[$3]+=$4} END{for(x in sum) {print x, sum[x] }}' inf_
traffic_ios.txt
output, 79893889
input, 59143208
```

说明如下。

- ❑ `/packets/`：通过模式匹配获得 `packets input` or `packets output` 的行（`packets` 后有一个空格）；
- ❑ `{sum[$3]+=$4}`：这里表示对第4列的数值进行累加，这个表达式也可以写成 `sum[$3]=sum[$3]+$4`。“+”的运算符和 C 语言中的一样。
- ❑ `END`：`END` 后是输出的结果。

大家也许会奇怪，这台设备的 `output` 为什么是大于 `input` 的。因为这是一台实验室的设备，平时几乎没有流量，接口的大部分流量主要来自路由协议的报文。通过这个方法来判断设备是否丢包并不是百分百准确的，不过在现网的环境中还是有一定的参考价值。当然，这个例子主要是为了说明 `awk` 的使用方法。

### 5.3.6 多文件操作

多文件的操作其实非常简单，只需要在最后加上多个文件名就可以，或者使用文件名的通配符。为了简单起见，这里复制了 `if_info_ios.txt` 文件，文件名为 `if_info_ios.txt.bak`。

```
$ awk '/Gi/ { inf[$5]++} END{print "down", inf["down"]; print "up", inf["up"]}'
if_info_ios.txt if_info_ios.txt.bak
```

```
down 10
up 14
```

或者

```
$ awk '/Gi/ { inf[$5]++} END{print "down", inf["down"]; print "up", inf["up"]}'
    if_info_ios.*
down 10
up 14
```

我们可以看到这次的结果刚好是之前的两倍，因为相同的内容被统计了两次。

awk 的功能是很丰富的，本节的内容远不能覆盖其所有的内容。但是，我们可以清楚地看到 awk 在处理文本表格的时候是非常方便的。网络设备 CLI 的输出中有很多内容都是通过文本表格的形式输出的。如，接口摘要信息、路由表信息、邻居信息（如 OSPF、ISIS、BGP、LLDP）等。这些信息都可以用 awk 进行一些简单的分析和处理。

无论是 grep 还是 awk，其主要功能是查询和统计，而下面介绍的 sed 则主要用在文本的流编辑上。

## 5.4 使用 sed 进行文本编辑

在对网络设备进行管理时，除了查询和统计之外也会涉及很多文本的修改和编辑工作，最常见的就是修改配置文件。本节将介绍如何通过 Sed 这个工具来编辑文件。

### 5.4.1 什么是 sed

sed 是一个非常老的文本处理工具，最早的版本可以追溯到 1974 年。我们现在在 Linux 平台上使用的 sed 是 GNU sed (<https://www.gnu.org/software/sed>)，当前版本是 4.4；而在 MAC OS X 平台上使用的是 BSD 版本的 sed。这两个版本存在细微的区别。本书使用 GNU 版本的 sed。

sed (stream editor) 是一种非交互式的基于流的编辑器。基于流的意思是文本会一行一行地经过 sed 的转换规则，然后将结果输出到标准输出上，而标准输出默认是屏幕。默认情况下，sed 并不会修改文本自身，而需要通过管道符“>”输出到另一个文件中；另外，也可以使用“-i”参数直接修改文本自身。由于 sed 的这个特点，其特别适合处理非常大的文本文件（如 GB 级的日志文件）或者进行有规则的文本修改，且适用于那些上下文无关的文本。

### 5.4.2 sed 语法简介

使用 sed 的命令如下：

```
$ sed [选项] '命令' 文件
```

说明如下。

- ❑ 选项是指 sed 的参数。
- ❑ 命令是指 sed 的命令集，这里的命令集有 25 个。
- ❑ 文件是指要处理的文件流。

sed 常用参数如表 5-8 所示。sed 常用命令如表 5-9 所示。

表 5-8 sed 常用参数

参数 (区分大小写)	说 明
-e	后面是对流内容 (通常是一行文本) 的操作方法。操作命令见表 5-9
-i	直接修改流文件的内容
-n	对不匹配的内容不进行输出
-f	指定 sed 的脚本文件

表 5-9 sed 常用命令

命令 (区分大小写)	说 明
d	删除行
i	在匹配行前加入文本内容
s/ 字符串 1/ 字符串 2/	用字符串 2 替换字符串 1
p	输出匹配的行
n	读取下一行的内容，并且用下一个命令来处理新的行
!	匹配的逆操作
a	在当前行后添加一行或多行内容

5.4.3 删除文件中的指定信息

sed 在处理网络设备的配置方面并不那么擅长，特别是强上下文关联的内容。在本节，我们会用到两个文件进行举例，这两个文件在 5.2.4 节中已经使用过，分别是 router\_isis\_ios.txt 和 router\_isis\_junos.txt。

【例 1】在 router\_isis\_junos.txt 文件中删除接口 ge-1/2/2.0 在 ISIS 中的配置。

```
$ sed -e '/ge-1\/2\/2\/d' router_isis_junos.txt
set interfaces ge-1/2/0 unit 0 description to-R1
set interfaces ge-1/2/0 unit 0 family inet address 10.0.0.2/30
set interfaces ge-1/2/0 unit 0 family iso
set interfaces ge-1/2/1 unit 0 description to-R2
set interfaces ge-1/2/1 unit 0 family inet address 10.0.1.2/30
set interfaces lo0 unit 0 family inet address 192.168.0.2/32
set interfaces lo0 unit 0 family iso address 49.0001.0192.0168.0002.00
set protocols isis interface ge-1/2/0.0
set protocols isis interface lo0.0
```



说明如下。

- ❑ -c: 见表 5-8。
- ❑ /ge-1/2/2/d: 由于接口名字包含了 “/” 符号, 因此, 使用 “\” 进行转义, 最后的 d 表示删除这行的内容。

【例 2】在文件 router\_isis\_ios.txt 中删除接口 ethernet0/0 中的 ip router isis 配置。

```
$ sed -e '/ethernet0\0/0/{n;d}' router_isis_ios.txt
router isis
    net 49.0001.0000.0000.0001.00
```

```
interface ethernet0/0
    ip address 172.17.1.1 255.255.255.0
```

```
interface ethernet0/1
    ip address 172.16.1.1 255.255.255.0
```

```
interface serial2/0
    ip router isis
    ip address 192.168.1.1 255.255.255.0
```

```
interface serial5/0
    ip address 172.21.1.1 255.255.255.0
```

说明如下。

- ❑ /ethernet0\0/0/: 匹配条件和例 1 类似。
- ❑ {n;d}: 这里是指, 当匹配成功后, 需要读下行的内容, 然后删除这行。



**注意** 这个命令在 MAC OSX 下不能正常工作, 因为 MAC OSX 使用的是 BSD sed, 而不是 GNU sed。

#### 5.4.4 在文件中进行查找替换

同时替换两个文件中的 ISO 地址中的 Area 信息, Area ID 由 49.0001 替换为 86.1234。命令如下:

```
$ sed -e 's/49.0001/86.1234/' router_isis_ios.txt router_isis_junos.txt
router isis
    net 86.1234.0000.0000.0001.00
<略>
interface serial5/0
    ip address 172.21.1.1 255.255.255.0

set interfaces ge-1/2/0 unit 0 description to-R1
<略>
```

```
set interfaces lo0 unit 0 family inet address 192.168.0.2/32
set interfaces lo0 unit 0 family iso address 86.1234.0192.0168.0002.00
set protocols isis interface ge-1/2/0.0
set protocols isis interface ge-1/2/2.0
set protocols isis interface lo0.0
```

说明：s/49.0001/86.1234/：使用查找替换的方式，见表 5-9 中的命令。

### 5.4.5 在文件中插入内容

在配置文件 router\_isis\_ios.txt 的接口 ethernet0/0 后面加上 shutdown 的命令。

```
$ sed -e '/ethernet0\/0/ a \ shutdown' router_isis_ios.txt
router isis
    net 49.0001.0000.0000.000b.00

interface ethernet0/0
    shutdown
    ip router isis
    ip address 172.17.1.1 255.255.255.0
<略>
interface serial5/0
    ip address 172.21.1.1 255.255.255.0
```

说明：a \ shutdown：命令 a 见表 5-9，表示在匹配文本后加入内容。因为 shutdown 前面有一个空格，因此使用“\”进行转义。

网络设备 CLI 输出的内容可以分为两大类：一类是设备的状态输出，通常为命令行执行的结果和系统日志；另一类是设备的配置。第一类通常以查询和统计为主，而第二类通常还有修改的需求。但是，大量的网络设备配置存在上下文相关性。因此，sed 并不太适合对配置文件进行修改。只有对上下文相关性要求小且又需要大规模修改时，sed 才会起到一定的作用。

## 5.5 文本编辑工具 vi 和 vim

网络设备的管理几乎是通过文本形式来操作的，因此必然会面临文本的编辑工作。本节会给大家介绍最常用的文本编辑软件 vi (visual interface)，它几乎是所有“黑客”使用的标准编辑器。

### 5.5.1 vi 和 vim 简介

vi 是 Linux/UNIX 平台下最常见的文本编辑器，其功能毫不逊色于很多图形化的编辑器。它可以对文本进行查找、删除、替换以及编辑，几乎可以涵盖日常的所有工作。使用

vi 并不需要图形界面的支持，只需要字符界面就可以了。vim 是 vi 的增强版，其实现在的 Linux 以及 MAC OS X 中的 vi 命令几乎都已经用 vim 进行了取代。在操作上，vim 和 vi 几乎一样。习惯了 vi 的使用，在使用 vim 的时候几乎没有什么变化。本节将不会刻意区分 vi 和 vim。下面给出 CentOS 7 和 MAC OS X 中 vi 命令的版本信息，我们可以看到，在这两个系统中 vi 命令已经替换成 vim 7.4。

```
Centos 7:
$ vi --version
VIM - Vi IMproved 7.4 (2013 Aug 10, compiled Jun 10 2014 06:56:12)
Included patches: 1-160
Modified by <bugzilla@redhat.com>
Compiled by <bugzilla@redhat.com>
<略>

MAC OSX Sierra:
$ vi --version
VIM - Vi IMproved 7.4 (2013 Aug 10, compiled Apr 4 2017 18:14:54)
Included patches: 1-898, 8056
Compiled by root@apple.com
<略>
```

在学习 vi / vim 时，vimtutor 是一个很好的入门教材。MAC OS X 系统默认带有这个教程，而且有中文版本。你可以在 MAC OSX 的命令行界面中输入：

```
$ vimtutor -g zh
```

如果在你的 CentOS 下没有这个命令，你可以通过如下命令来安装。

```
$ sudo yum -y install vim-enhanced
$ vimtutor -g zh #然后使用这个命令进行查看。
```

图 5-2 是 vimtutor 教程。这个教程既可以当作初学者的教程，也可以作为速查手册供相关人员使用。

### 5.5.2 vim 编辑器的模式

初学者会觉得 vim 很不好用，因为它和我们在图形化界面中使用的文本编辑器有很多的区别，比如，没有菜单可以选择，不能直接编辑文件内容，不能直接用鼠标选择内容等，而且很多功能是通过很多命令来实现的。因此，对 vim 的学习曲线，一开始可能是比较陡峭的。但是，一旦你跨过难度曲线的高点，那么使用 vim 进行文本编辑还是挺方便的。

vim 的文档（<http://www.vim.org/docs.php>）中提到，vim 有六个基本模式（basic mode）和六个额外模式（additional mode）。我们常用的是三个模式，分别是普通模式、插入模式和命令行模式。图 5-3 是 vim 模式之间的关系。



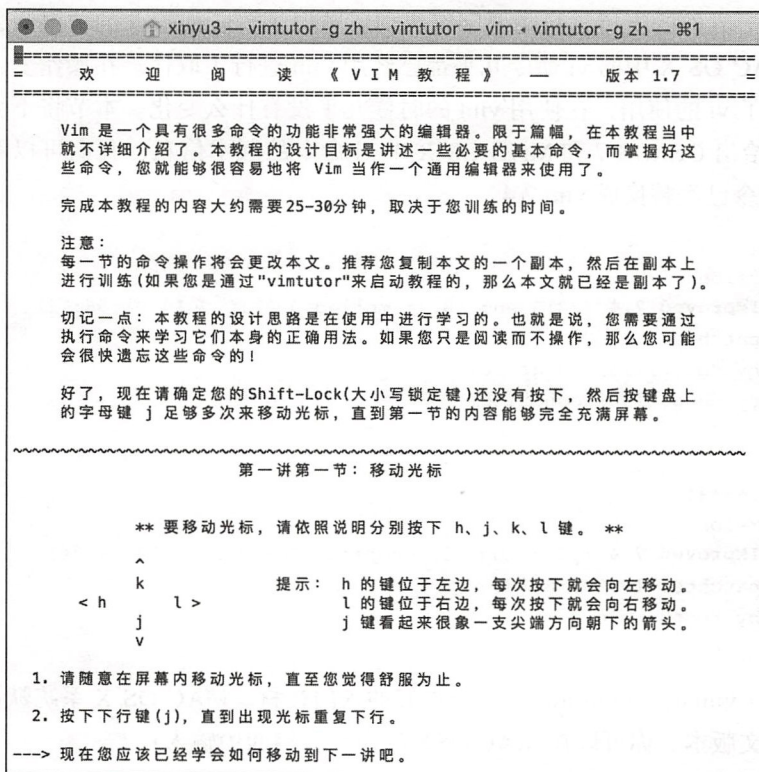


图 5-2 vimtutor 教程

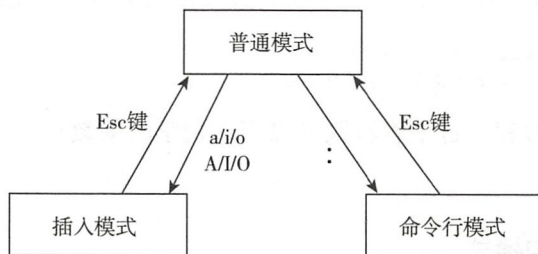


图 5-3 vim 模式之间的关系

### 1. 普通模式

当打开文档的时候，默认会进入普通模式（normal mode）。通过这个模式可以进入其他模式。在这个模式下，我们可以移动光标、删除文本及复制、粘贴文本。这些都是通过按键命令来完成的。表 5-10 给出了 vim 普通模式下的常用命令，这里列出的命令主要是光标移动命令。如果觉得这些命令一时难以记忆，那么使用方向键也可以实现相同功能。表 5-11 列出了部分 vim 普通模式下的常用命令，这里引出的命令主要是删除字符和撤销等命令。

表 5-10 vim 普通模式下光标移动命令

命令 (区分大小写)	说 明
h 或者是左方向键	光标向左移动一个字符
l 或者是右方向键	光标向右移动一个字符
j 或者是下方向键	光标向下移动一个字符。和向下移动一行效果相当
k 或者是上方向键	光标向上移动一个字符。和向上移动一行效果相当
数字 0 或 ^ (数字 6 上的字符)	移动光标到行首, 在正则表达式中 “^” 也表示行首
\$ (数字 4 上的字符)	移动光标到行尾, 在正则表达式中 “\$” 也表示行尾
G	移动光标到文件最后
gg	移动光标到文件开头
H	移动光标到当前屏幕的第一行
M	移动光标到当前屏幕的中间一行
L	移动光标到当前屏幕的最后一行
w	移动光标到下一个单词的词首
b	移动光标到上一个单词的词首
e	移动光标到下一个单词的词尾

表 5-11 vim 普通模式下其他命令

命令 (区分大小写)	说 明
n dd	n 是数字, 可以是多位数。表示删除光标所在的行和后续的行的内容。如果没有数字, 则 n 默认等于 1, 即只删除当前行
dw	删除光标往后的一个单词以及单词后的空格
D 或者 d\$	删除从光标到本行最后的文本
X	向前删除一个字符
x	删除光标所在位置的一个字符
u	撤销操作, 需要多次撤销动作, 则多次输入 u
r	替换光标所在位置的字符

2. 插入模式

插入模式主要用于输入文本内容。表 5-12 列出了进入插入模式的常用命令。熟悉这些命令可以提高 vi 的使用效率。

3. 命令行模式

在 vim 编辑器的最下面输入命令, 从而来操作文本内容或改变 vim 环境的模式称为命令行模式。注意其与普通模式的区别, 在普通模式下, 先输入冒号 “:”, 然后输入相应的命令。为了便于说明, 笔者把命令行模式常用命令分为四类 (vim 的命令其实很多, 这里只

列出一些常用的命令)，如表 5-13 所示。

表 5-12 进入插入模式的常用命令

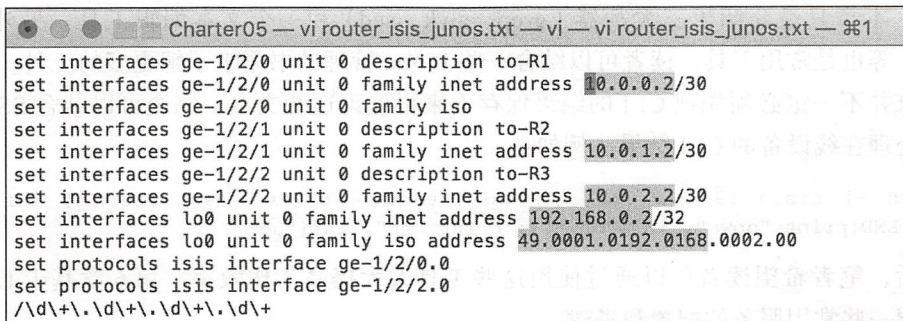
命令 (区分大小写)	解 释
i	在当前光标前插入文本内容
I	在当前光标所在行的开头插入文本内容
o	在当前光标所在行的下一行插入新行，并插入文本内容
O	在当前光标所在行的上一行插入新行，并插入文本内容
a	在当前光标后插入文本内容
A	在当前光标所在行的结尾插入文本内容
C	删除光标所在位置到行结尾处所有的字符，并插入文本内容

表 5-13 命令行模式常用命令

命令分类	命令 (区分大小写)	解 释
查找	/word	从当前光标开始查询 “/” 后的字符串的内容。这个 word 可以是一个正则表达式（这里的表达式和本章开始提到的正则表达式会有一点点小的区别，大家可以参考 <a href="http://vimregex.com">http://vimregex.com</a> ）。这个命令可以先输入 “:”，也可以不输入，在普通模式下直接输入 “/” 也是可以的
	n 和 N	在进行查找时，如果有多个匹配项目，输入 n 可以查找下一个匹配项，输入 N 可以查找上一个匹配项
	:set hlsearch :set nohlsearch	设置或取消对查找匹配项的高亮显示，见图 5-4
替换	:n1,n2s/word1/word2/g	在第 n1 行到第 n2 行之间查找匹配 word1 的内容并且替换为 word2 的内容。这里的 “/” 可以替换为 “#” 字符，这对需要查找和替换的内容中出现 “/” 字符的情况是很有用的，见图 5-5
	:1,\$s/word1/word2/gc 或 :%s/word1/word2/gc	“%” 同 “1,\$”，都是全文替换。字母 g 后的 c 是指，每次遇到需要替换的内容时需要得到用户的确认
保存与退出	:w	保存文件不退出
	:w!	如果文件的属性是只读，则强制保存
	:wq	保存文件并退出
	:q	退出。如果文件有修改，则无法退出
	:q!	放弃修改，直接退出
	ZZ	如果文件修改，则保存退出；如果没有修改，则直接退出
	:n1,n2 w [ 文件名 ]	把文本中第 n1 行到第 n2 行的内容保存到另外一个文件中
其他	: set number 或 :set nu :set nonumber 或 set nonu	显示或取消行号的显示

图 5-4 和图 5-5 分别给出了两个简单的例子。其中，图 5-4 是使用 vim 进行 IP 地址查找的例子，在 Mac OS X 中，我们可以看到，被查找到的文本内容被“染”成了其他颜色。





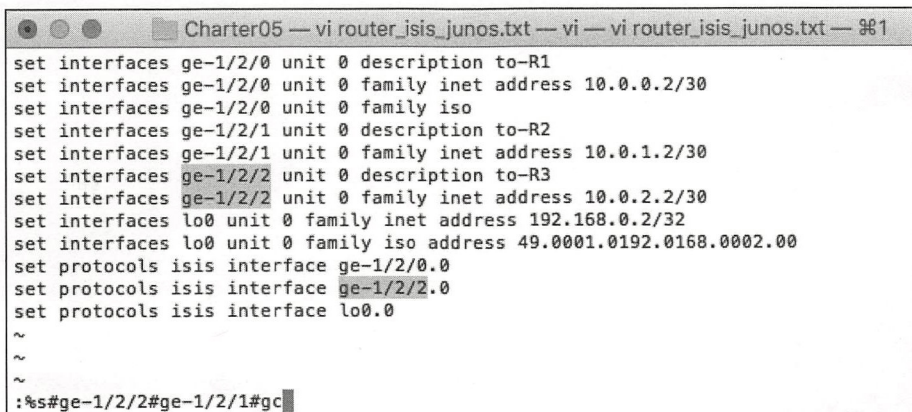
```

Charter05 — vi router_isis_junos.txt — vi — vi router_isis_junos.txt — %1
set interfaces ge-1/2/0 unit 0 description to-R1
set interfaces ge-1/2/0 unit 0 family inet address 10.0.0.2/30
set interfaces ge-1/2/0 unit 0 family iso
set interfaces ge-1/2/1 unit 0 description to-R2
set interfaces ge-1/2/1 unit 0 family inet address 10.0.1.2/30
set interfaces ge-1/2/2 unit 0 description to-R3
set interfaces ge-1/2/2 unit 0 family inet address 10.0.2.2/30
set interfaces lo0 unit 0 family inet address 192.168.0.2/32
set interfaces lo0 unit 0 family iso address 49.0001.0192.0168.0002.00
set protocols isis interface ge-1/2/0.0
set protocols isis interface ge-1/2/2.0
/d\+\.\d\+\.\d\+\.\d\+

```

图 5-4 vim 查询高亮显示

图 5-5 是一个文本替换的例子，这里把 ge-1/2/2 替换为 ge-1/2/1。



```

Charter05 — vi router_isis_junos.txt — vi — vi router_isis_junos.txt — %1
set interfaces ge-1/2/0 unit 0 description to-R1
set interfaces ge-1/2/0 unit 0 family inet address 10.0.0.2/30
set interfaces ge-1/2/0 unit 0 family iso
set interfaces ge-1/2/1 unit 0 description to-R2
set interfaces ge-1/2/1 unit 0 family inet address 10.0.1.2/30
set interfaces ge-1/2/2 unit 0 description to-R3
set interfaces ge-1/2/2 unit 0 family inet address 10.0.2.2/30
set interfaces lo0 unit 0 family inet address 192.168.0.2/32
set interfaces lo0 unit 0 family iso address 49.0001.0192.0168.0002.00
set protocols isis interface ge-1/2/0.0
set protocols isis interface ge-1/2/2.0
set protocols isis interface lo0.0
~
~
~
~
:%s#ge-1/2/2#ge-1/2/1#gC

```

图 5-5 vim 文本替换

vim 是一个功能十分丰富的文本编辑器，甚至很多人把 vim 作为编程用的 IDE (Integrated Development Environment, 集成开发环境)。vim 官网将 vim 定位为一个“开发工具”，而不仅仅只是一个简单的文本处理工具。本节介绍的 vi/vim 的功能只是其常用的一些功能，更加深入的内容可以参考其他内容。

一个熟练的 vim 使用者，使用 vim 就像是“手指的舞蹈”。希望读者也能在键盘上跳出优美的舞蹈。

## 5.6 小结

传统网络设备 CLI 的输出都是纯文本。管理网络设备大部分时间是管理、分析和处理这些文本内容。对于一个 Linux 或 UNIX 的系统管理员而言，其日常的工作是处理大量的文本内容，如系统的配置文件、某个服务的配置文件或者各种日志文件等。另外，这些文本的格式和网络设备 CLI 输出的格式是非常相似的。Linux 或 UNIX 有很多非常好用的文

本工具，本章只是介绍了三大利器（grep、awk、sed）和一个文本编辑工具。另外，cut、sort、wc 等也是常用工具，读者可以结合一些 Linux 管理的书籍来了解更多的工具。

大家并不一定必须先把 CLI 的结果保存下来再处理这些文本，完全可以结合 SSH 以及管道来处理在线设备的 CLI 结果。例如：

```
$ ssh -l cisco 192.168.0.2 show ip interface brief | awk '/Gi/ { inf[$5]++;  
END{print "down", inf["down"]; print "up", inf["up"]}'
```

最后，笔者希望读者可以通过使用这些工具大大提高工作效率。第 6 章基于 Linux 的环境介绍一些常用服务的配置和搭建。

## 常用基础服务搭建

在网络运维和建设中经常需要用到一些基础服务，比如网络设备进行软件升级时，需要使用 TFTP 服务来传送网络设备的操作系统。假设我们在 DC（Data Center，数据中心）中使用了 ZTP 进行网络设备自动化上线部署，还会用到 DHCP 服务。在本章，我们会介绍以下几种服务的搭建及其基本功能的配置。

□ TFTP;

□ DNS;

□ DHCP。

网络工程师并不像系统管理员，大家搭建这些基础服务基本都是为了满足网络设备的相关需求，如设备上线部署、后期自动化运维等方面。对这些基础服务的需求也往往都是功能上的满足，而非性能和容量需求。但是，如果为了满足云中虚拟网元节点（纯软件网元、NFV 节点）或者是大型网络的需求，那么对性能和容量的要求则会有较大提高。不过这也是建立在实现基本功能基础上的。因此，本章会先关注功能上的实现。

对于网络工程师而言，使用这些服务是主要的。因此，如何简单、快速地部署相关服务应用是首要的。本章所有的相关服务都会采用 Docker 的方式来进行部署。其原因有以下几点。

首先，Docker 对服务的部署会相对简单。网络工程师并不需要和系统管理员一样从源代码编译可执行文件开始，而是通过简单的方式来搭建这些基础服务。

其次，Docker 可对服务进行模块化处理，便于网络工程师进行快速的迁移服务。现在除了传统的 Linux 的一些服务组件可以 Docker 化（容器化），越来越多的网络设备厂家也推出了基于 Docker 的产品，比如 Juniper 发布的 cSRX，Cisco 也有很多 Docker 发布的产品。除了基于 Docker 发布的产品，传统的物理交换机或路由器也有一些平台支持 Docker 服务，



即可以在这些传统的网络设备上运行 Docker 镜像。比如, Arista 的 EOS 平台就可以直接运行 Docker 镜像。这里笔者就不一一举例了, 读者可以通过互联网了解各个网络厂商对 Docker 的支持情况。

最后, 管理虚拟机或 Docker 的网络也许是网络工程师今后需要接触的内容。采用这种方式对网络基础服务进行部署可以拓宽大家的网络知识边界。

## 6.1 Docker 基础

Docker 这个名词对于大部分网络工程师而言应该不会那么陌生, 但至于它具体是什么、能干什么以及怎么使用, 大家也许就不一定很清楚了, 尤其是对于那些管理与维护传统网络的工程师而言。希望通过本节介绍达成以下几个目标:

- ❑ 了解什么是 Docker;
- ❑ 如何使用已有的 Docker 镜像快速部署应用(服务);
- ❑ 构建自定义的简单 Docker 镜像。

本章将不会涉及如何运维和管理 Docker 网络的相关内容, 仅把 Docker 作为一个应用工具介绍其使用。

### 6.1.1 什么是 Docker

Docker 是一种容器技术, 即将单个操作系统的资源划分到一些独立组的技术。和虚拟化相比, 这样的技术不需要实现指令级的模拟, 也不需要即时编译过程。容器在提供隔离的同时, 通过共享底层的资源来节省系统的开销, 因此容器比虚拟化的开销要小很多。表 6-1 提供了容器和虚拟机的比较。如果用网络工程师比较熟悉的知识领域做一个类比: 利用光传输的波分系统构建一个网络与利用 MPLS VPN 技术构建一个网络, 前者可以类比为虚拟化, 而后者就是容器。容器可以运行在物理服务器上, 也可以运行在虚拟化后的虚拟机中, 就好像 MPLS VPN 网络既可以构建在裸光纤上, 也可以构建在波分系统上(虽然这个类比也许不是那么恰当, 但是, 希望通过这样的类比给网络工程师以一个更加直观的方式来认识它)。

表 6-1 容器与虚拟机的比较

特 性	容 器	虚拟机
启动时间	秒级	分钟级
体积	通常为 MB	通常为 GB
性能	接近原生系统	弱于原生系统
系统支持的数量	单机支持上千个容器	通常为几个到几十个
对应用的支持	容器内为 Linux (已经有 Windows 容器项目来支持 Windows 应用)	虚拟机可以是任何系统

容器这种技术形式已经存在一段时间，如 BSD Jails（2000 年随着 FreeBSD 4.0 发布）以及 Solaris Zones（2005 年随着 Solaris 10 发布）。在 Linux 平台中，容器技术出现之前也有类似的技术，如 Linux Vserver（<http://linux-vserver.org>）和 OpenVZ（<https://openvz.org>）。Linux 容器技术通常被认为（也存在一些争议，我们姑且也这样认为）是 2008 年在 Linux 内核中加入了内核命名空间（kernel namespaces）和用户命名空间（user namespaces），在 2014 年发布了 LXC 1.0.0 版本（<https://linuxcontainers.org>）。

Docker 于 2013 年发布，使用 Go（Google 公司推出的语言，也称 Golang）语言，基于 Linux 内核的 CGroup<sup>①</sup>、namespace<sup>②</sup>以及 Union FS<sup>③</sup>等技术，对进程进行了隔离，它是一种操作系统层面的虚拟化技术。现在 Docker 软件主要有两个版本：一个是 Docker CE（社区版本），这是一个免费的版本；另一个是 Docker EE（企业版本），这是一个付费的版本。Docker 除了这两个以 Docker 命名的版本，还有一个开源的版本——Moby（<https://mobyproject.org>）。本章使用的是 Docker CE 版本。Docker 基于容器的技术实现了文件系统、网络以及进程的隔离，并且极大简化了容器在创建和运维的工作，使得它比虚拟机更加轻量 and 快捷。作为网络工程师，我们可以先关注于它的使用，利用 Docker 这个工具快速地构建我们的基础服务。下面先介绍 Docker 的三个基本概念。

### 6.1.2 Docker 的基本概念

Docker 包含如下三个基本概念：

- ❑ 镜像（image）；
- ❑ 容器（container）；
- ❑ 仓库（repository）。

这三个基本概念对我们使用 Docker 很有帮助，因为它们贯穿了 Docker 的整个生命周期。

#### 1. 镜像

镜像是一些只读层的统一名称，是一个比较特殊的文件系统。这个文件系统通常包括运行需要的程序、库文件以及配置文件。通常镜像不包含动态的数据，换言之，在创建镜像的时候，这些内容就被固化下来了，无论这个镜像被启动了或者停止了多少次，其内容都是不变的。Docker 使用了 AUFS（Advanced Multi-layered Unification Filesystem，高级多层统一文件系统）技术，其中的文件是通过多层的文件系统联合组成的。在构建镜像的时候会一层一层地进行构建，前一层是后一层的基础。每一层构建完成后就不会发生变化。AUFS 的结构示意图见图 6-1。

① <https://zh.wikipedia.org/wiki/Cgroups>。

② [https://en.wikipedia.org/wiki/Linux\\_namespaces](https://en.wikipedia.org/wiki/Linux_namespaces)。

③ [https://en.wikipedia.org/wiki/Union\\_mount](https://en.wikipedia.org/wiki/Union_mount)。



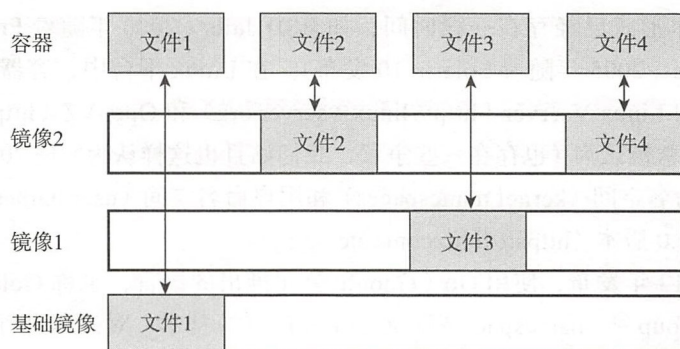


图 6-1 AUFS 的结构示意图

我们可以通过一个类比来进行解释：假设我们需要建立一个大型的 MPLS VPN 网络，配置的结构如图 6-2 所示。

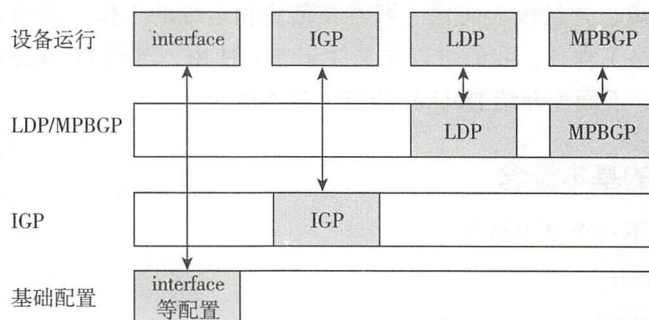


图 6-2 MPLS VPN 配置结构

首先，我们需要配置每台路由器接口的 IP 地址和接口的 MPLS 能力，这些配置相当于最底下的一层配置信息。

其次，我们需要配置 IGP 路由协议，用于传递设备的路由信息，这一层可以理解为第二层的配置。

再次，我们需要配置一个标签分配协议，假设我们使用了 LDP 协议。

最后，我们也许还需要配置 BGP 的协议，用于传递业务路由信息（假定这个 BGP 的配置已经包含 MP-BGP VPNv4 的配置），这一层就是最上层。



**注意** 在大部分网络设备上，并不能在设备配置上体现出层次结构，配置完成后都位于一个配置文件中。这和 AUFS 的文件系统一样，在你使用的时候并不会觉得在分层的文件系统。这种层次的划分只是在网络工程师对网络的设计思路中。

另外，Cisco IOS XR 和 Juniper JUONS 的配置中都有 group 的配置方式，可以把一组配置通过 group 命令组合在一起，然后通过 apply-group 的方式应用到不同的配置模块中。通过这种方式，我们可以很灵活地组织设备的配置。这部分的内容可以参



考以下资料。

① Cisco IOS XR。

Configuring Flexible Command Line Interface Configuration Groups

[https://www.cisco.com/c/en/us/td/docs/routers/crs/software/crs\\_r4-3/system\\_management/configuration/guide/b\\_sysman\\_cg43crs/b\\_sysman\\_cg43crs\\_chapter\\_010001.html](https://www.cisco.com/c/en/us/td/docs/routers/crs/software/crs_r4-3/system_management/configuration/guide/b_sysman_cg43crs/b_sysman_cg43crs_chapter_010001.html)

② Juniper JUNOS。

Understanding Junos OS Configuration Groups

[https://www.juniper.net/documentation/en\\_US/junos/topics/concept/junos-software-configuration-groups-understanding.html](https://www.juniper.net/documentation/en_US/junos/topics/concept/junos-software-configuration-groups-understanding.html)

每一层都依赖于下面一层，修改下层会导致上层无法正常工作。不过，在网络设备配置的这个例子中，每一层都不是固化的，是可以被修改的。但是，Docker 的每一层都是固化的，是不可修改的。每次运行 Docker 镜像的时候，镜像的最顶层会添加一个可写写的层。文件系统发生变化，只会影响最上面的一层。如果这个运行环境被保存，那么这一层的文件系统会保持在最上层。层与层之间都会保留区别，并且还会计算一个哈希值。这种设计模式是很容易实现文件系统的回滚操作的。对于网络工程师而言，我们是在 NETCONF 协议中较多地接触回滚概念的。NETCONF 协议定义了很多能力（capability），其中两个是候选（candidate）配置能力和回滚（rollback）能力。当我们开始修改设备配置时，网络设备的操作系统会把现有的所有配置复制一份到内存中，这个配置可以任意修改。但是，在修改这个配置的时候，设备正在运行的配置是不变的。这种方式和 Docker 镜像文件系统有一些类似。我们对网络设备进行配置提交（commit 操作）后，网络设备通常会保留上一次的配置和本次修改完的配置信息。这样的形式非常类似 Docker 层的概念。

## 2. 容器

容器的本质是进程，但是它和宿主机的进程有所不同，容器的进程是运行在自己单独的命名空间里面的。每个容器都有自己的网络、文件系统以及进程空间。容器还具有容器存储层，这一层是具有读写能力的层。容器的存储层具有和容器一样的生命周期，当容器停止的时候，这一层也就消失了（前提是没有保存）。按照 Docker 的最佳实践，对于需要保存的数据，我们需要把这些数据保存到数据卷中。本章后续章节将使用数据卷来保存服务的配置文件和一些数据。

这里，我们继续使用镜像中提到的例子，假设我们需要在 MPLS VPN 网络中增加一个 MPLS VPN 网络，那么我们需要在某些 PE 设备上创建一些 VRF（BGP 的配置假设已经完成且不需要修改）。这些新加的配置可以理解为容器的存储层。当这个 MPLS VPN 消亡的时候，我们需要删除这些相关的配置（通常是 PE 上的 VRF 配置）。另外，每个 MPLS VPN 中还会有 VPN 内的路由信息，而这些信息往往是不保存的。这些信息其实也可以算作 MPLS VPN 的内容。而 VRF 其实就是路由器上的一个进程。

### 3. 仓库

当我们在一台主机上构建了一个镜像时，我们当然希望它可以被其他主机使用。那么我们就需要把镜像集中地存放到一个地方，而提供这个服务的就是仓库。仓库既有公有的，也有私有的。较为常用的公有仓库是 <https://hub.docker.com>，这里有非常丰富的 Docker 镜像。企业也可以构建自己的私有仓库。如何构建自己的私有仓库，读者可以参考 <https://docs.docker.com/registry/deploying>。

## 6.1.3 Docker 的运行环境

由于 Docker 是基于 Linux 的容器技术，因此它可以完美地运行在 Linux 操作系统上。随着技术的发展，Docker 也可以运行在 MAC OS X 和 Microsoft Windows 环境中。下面分别介绍在 CentOS 和 Ubuntu 环境下如何构建 Docker 的运行环境。

### 1. CentOS 系统

首先，对于 CentOS，我们推荐使用 64 位的 CentOS7 系统，安装一些基本的工具。

```
$ sudo yum install -y yum-utils device-mapper-persistent-data lvm2
```

其次，添加 Docker CE（本书以 CE 版本为例，关于 Docker 的版本可以参考本书 6.1.1 节的内容）的 yum 仓库。

```
$ sudo yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
```

然后，安装 Docker CE。

```
$ sudo yum install docker-ce
```

最后，启动 Docker 服务

```
$ sudo systemctl start docker
```

### 2. Ubuntu 系统

首先，对于 Ubuntu 系统，我们推荐使用 64 位系统，版本可以是 14.04、16.04 以及 17.04。这里以 16.04 为例，先添加 Docker 的 APT 软件库。

添加 Docker 官方的 GPG 公钥：

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
公钥的指纹信息为 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88
$ sudo apt-key fingerprint 0EBFCD88
```

```
pub 4096R/0EBFCD88 2017-02-22
    Key fingerprint = 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88
uid Docker Release (CE deb) <docker@docker.com>
sub 4096R/F273FCD8 2017-02-22
```



添加软件库：

```
$ sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
```

其次，开始安装 Docker CE 软件版本。

更新软件：

```
$ sudo apt-get update
```

安装 Docker CE 软件：

```
$ sudo apt-get install docker-ce
```

和 CentOS 不同的是，Ubuntu 安装 Docker CE 后，自动添加并启动了 Docker 服务。

### 6.1.4 启动 Docker 镜像

无论什么版本的 Linux，其安装好 Docker 后的操作几乎是一样的。我们可以先启动一个 hello world 的 Docker 镜像。

```
$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b04784fba78d: Pull complete
Digest: sha256:f3b3b28a45160805bb16542c9531888519430e9e6d6fffc09d72261b0d26ff74f
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to
   your terminal.
```

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

```
https://cloud.docker.com/
```

For more examples and ideas, visit:

```
https://docs.docker.com/engine/userguide/
```

我们可以看到，Docker 会从 [hub.docker.com](https://hub.docker.com) 下载需要的镜像文件。在上面的这个例子



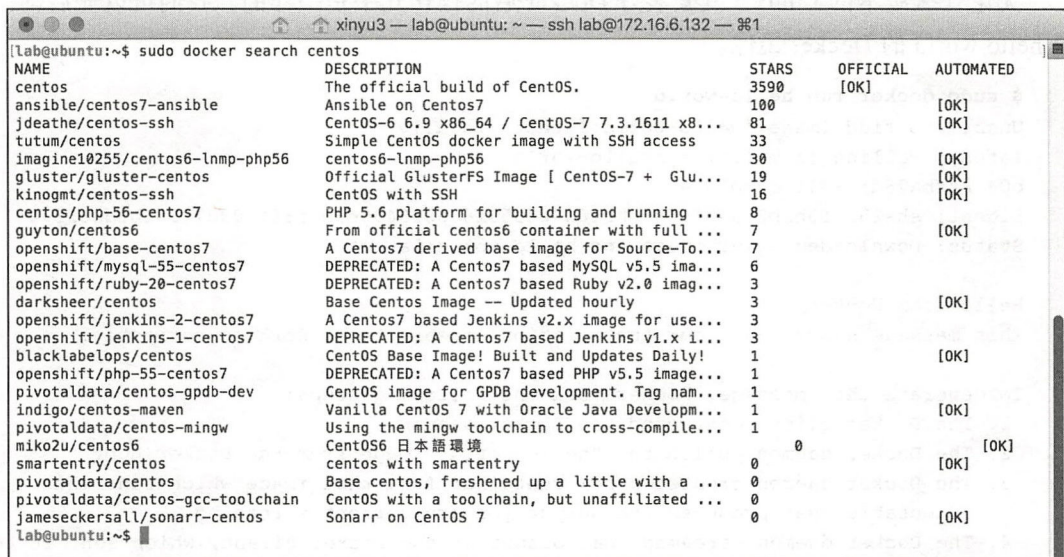
中, hello-world: latest 就是镜像名了。下载后, Docker 会运行这个进程。当本地没有你需要的 Docker 镜像文件时, Docker 会从默认仓库下载镜像文件, 然后运行它。镜像文件运行结束后退出 Docker 进程。这个例子使用了最简单的方式启动了一个 Docker。

### 6.1.5 构建 Docker 镜像

6.1.4 节使用的镜像是从 hub.docker.com 自动下载来的。但在很多时候, 公有仓库的镜像并不一定能满足大家的需要。这时需要我们自己来定制自己需要的 Docker 镜像文件。对镜像的定制通常有两种方式, 一种方式是在现有的 Docker 镜像基础上, 通过手动修改后使用 commit 命令来保存镜像; 另一种方式是通过 Dockerfile 文件来构建镜像。下面通过两个简单的例子来演示其过程。

#### 1. 通过 commit 命令来保存镜像

首先, 从公有仓库下载一个需要修改的定制容器。比如我们可以基于 CentOS 来定制镜像, 我们可以使用 Docker 命令来查找 CentOS 相关的镜像 (见图 6-3)。



NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
centos	The official build of CentOS.	3590	[OK]	
ansible/centos7-ansible	Ansible on Centos7	100		[OK]
jdeathe/centos-ssh	CentOS-6 6.9 x86_64 / CentOS-7 7.3.1611 x8...	81		[OK]
tutum/centos	Simple CentOS docker image with SSH access	33		
imagine10255/centos6-lamp-php56	centos6-lamp-php56	30		[OK]
gluster/gluster-centos	Official GlusterFS Image [ CentOS-7 + Glu...	19		[OK]
kinogmt/centos-ssh	CentOS with SSH	16		[OK]
centos/php-56-centos7	PHP 5.6 platform for building and running ...	8		
guyton/centos6	From official centos6 container with full ...	7		[OK]
openshift/base-centos7	A Centos7 derived base image for Source-To...	7		
openshift/mysql-55-centos7	DEPRECATED: A Centos7 based MySQL v5.5 ima...	6		
openshift/ruby-20-centos7	DEPRECATED: A Centos7 based Ruby v2.0 imag...	3		
darksheer/centos	Base Centos Image -- Updated hourly	3		[OK]
openshift/jenkins-2-centos7	A Centos7 based Jenkins v2.x image for use...	3		
openshift/jenkins-1-centos7	DEPRECATED: A Centos7 based Jenkins v1.x i...	3		
blacklabelops/centos	CentOS Base Image! Built and Updates Daily!	1		[OK]
openshift/php-55-centos7	DEPRECATED: A Centos7 based PHP v5.5 image...	1		
pivotaldata/centos-gpdb-dev	CentOS image for GPDB development. Tag nam...	1		
indigo/centos-maven	Vanilla CentOS 7 with Oracle Java Developm...	1		[OK]
pivotaldata/centos-mingw	Using the mingw toolchain to cross-compile...	1		
miko2u/centos6	CentOS6 日本語環境	0		[OK]
smartentry/centos	centos with smartentry	0		[OK]
pivotaldata/centos	Base centos, freshened up a little with a ...	0		
pivotaldata/centos-gcc-toolchain	CentOS with a toolchain, but unaffiliated ...	0		
jameseckersall/sonarr-centos	Sonarr on CentOS 7	0		[OK]

图 6-3 在 Docker 仓库中查找 CentOS

然后, 使用图 6-4 中的命令启动 CentOS。我们可以看到, 由于第一次运行这个镜像, Docker 会下载 CentOS 镜像文件。其中, -i 表示让容器一直保持打开状态, -t 表示让 Docker 分配一个伪终端, CentOS 表示运行的镜像名, /bin/bash 表示启动容器后运行的命令 (这里的命令表示打开一个 bash 的终端)。

```
$ sudo docker run -i -t centos /bin/bash
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
```

```

74f0853ba93b: Pull complete
Digest: sha256:26f74cefad82967f97f3eeef88c1b6262f9b42bc96f2ad61d6f3fdf544759b8
Status: Downloaded newer image for centos:latest
[root@3e674bf610d6 /]#

```

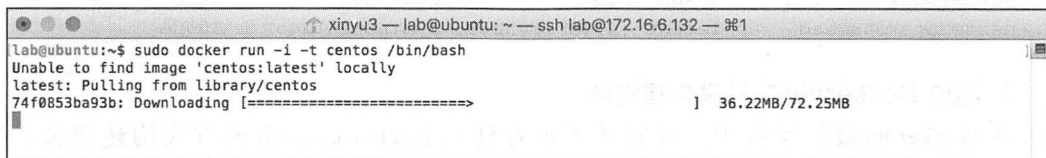


图 6-4 运行 CentOS 镜像

下载完成后，我们可以看到这个容器已经运行，并且我们可以看到 `/bin/bash` 已经运行了。

接下来，我们就可以在这个容器中安装或设置我们需要的内容。假设，我们这次需要安装 `dnsmasq` 这个软件（这个软件会在本章的后续章节中多次出现，后面我们会介绍这个软件的一些情况）。

```

[root@3e674bf610d6 /]# yum install -y dnsmasq
Loaded plugins: fastestmirror, ovl
<略>
Running transaction
  Installing : systemd-sysv-219-30.el7_3.9.x86_64                1/2
  Installing : dnsmasq-2.66-21.el7.x86_64                      2/2
  Verifying  : systemd-sysv-219-30.el7_3.9.x86_64              1/2
  Verifying  : dnsmasq-2.66-21.el7.x86_64                      2/2
Installed:
dnsmasq.x86_64 0:2.66-21.el7
Dependency Installed:
systemd-sysv.x86_64 0:219-30.el7_3.9
Complete!

```

到目前为止，我们已经安装好了 `dnsmasq` 这个软件，现在我们希望把它的状态保存下来，这样在下次运行的时候就不用再安装一次 `dnsmasq` 的软件了。为了实现这个目的，我们需要先使用 `exit` 命令退出当前的容器，然后使用 `docker commit` 命令。

```

[root@3e674bf610d6 /]# exit
exit
lab@ubuntu:~$ sudo docker commit 3e674bf610d6 netdevops/dnsmasq
[sudo] password for lab:
sha256:494db1f05b691d2ccb5a62593801e17362a6f1b94477fc7ac6cd7a4de0624c6a

```

命令 `commit` 后面的 ID 是前面运行 CentOS 容器的 ID，这个 ID 就是我们看到的 CentOS 容器的主机名。当然正确的做法是通过命令 `docker ps -a` 来查看。

```

$ sudo docker ps -a
CONTAINERID  IMAGE      COMMAND             CREATED          STATUS          PORTS          NAMES
3e674bf610d6 centos     "/bin/bash"        28 minutes ago  Exited (0)     3 minutes go

```



commit 保存后，我们可以通过命令 `docker images` 来查看。这里的镜像就是我们通过手动方式来创建的一个基于 CentOS 并安装了 `dnsmasq` 软件的镜像。

```
$ sudo docker images
REPOSITORY      TAG    IMAGE ID          CREATED          SIZE
netdevops/dnsmasq  latest  494db1f05b69    5 minutes ago   300MB
```

## 2. 通过 Dockerfile 文件来构建镜像

在 Docker 的最佳实践中，其实并不推荐使用 Docker commit 的方式构建镜像。因为，使用这样的构建方式，我们并不清楚这个镜像到底包含了什么内容。我们推荐使用 Dockerfile 的定义文件以及使用 Docker build 命令来构建镜像。接下来我们将使用 Dockerfile 文件来构建一个与上面例子一样的 Docker 镜像。

我们需要创建一个目录并在目录中创建一个名字为 Dockerfile 的文件（注意文件名的大小写）。

```
$ mkdir dnsmasq
$ cd dnsmasq
$ touch Dockerfile
```

现在我们可以使用 VIM 来编辑 Dockerfile 文件。文件内容如下：

```
# Version: 1.0.0
FROM centos:latest
MAINTAINER Xin Yu "yuxin@netdevops.cn"
RUN yum install -y dnsmasq
EXPOSE 53 53/udp
ENTRYPOINT ["dnsmasq", "-k"]
```

Dockerfile 文件是由一组命令和参数组成的。每条命令都必须大写，命令后就是参数部分。这些命令会从上到下依次执行，每执行一次命令将会形成单独的镜像层。通常 Dockerfile 中的流程如下（下面的过程都是自动完成的）：

- ❑ 从一个基础镜像运行一个容器。
- ❑ 执行一条命令修改容器中的内容。
- ❑ 执行 Docker commit 操作，从而完成一个新的镜像层的提交。
- ❑ 基于刚刚提交的镜像运行一个新的容器。
- ❑ 执行 Dockerfile 中的下一条命令，直到所有命令完成。

每一个 Dockerfile 文件都是从 FROM 命令开始的。FROM 命令指定的是一个本地存在或者是在 `hub.docker.com` 上存在的一个镜像。这个镜像是后续内容的基础。在上面的例子中，我们使用了 `centos:latest`。

接下来是 MAINTAINER 命令。这条命令会告诉大家这个镜像的维护者是谁。当然，这条命令不是必需的。不过，从可维护性来说，最好能添加这条命令。另外，在 Dockerfile 的开头可以添加更多描述性文字，上面的例子在开头只给出了版本信息。



下面是一个 RUN 命令。RUN 命令会在当前的镜像中运行指定的操作。上面的例子使用 yum 来完成安装 dnsmasq 的操作。默认情况下，RUN 后面的操作会在容器的 /bin/sh -c 后执行。

再下一行是 EXPOSE 命令。这个命令可以告诉 Docker 在运行这个镜像的时候，在容器中会打开什么端口。在上面的例子中，我们指定了 TCP 53 和 UDP 53 端口。EXPOSE 后面默认指定的是 TCP 端口，如果是 UDP 端口就需要明确写出来，就像 53/udp。

最后一行是 ENTRYPOINT 命令。这个命令和 EXPOSE 一样，不是在构建镜像的时候起作用的，它们都是在镜像被运行时、加载容器时起作用的。这个命令后的内容是 Docker 在运行时需要执行的容器内的命令。在上面的例子中，容器运行的时候将会执行 dnsmasq -k 让 dnsmasq 这个服务。

在编写 Dockerfile 后，我们需要通过 docker build 命令来完成镜像的构建。

```
$ sudo docker build -t="netdevops/dnsmasq" .
Sending build context to Docker daemon 2.048kB
Step 1/5 : FROM centos:latest
----> 328edcd84f1b
Step 2/5 : MAINTAINER Xin Yu "yuxin@netdevops.cn"
----> Running in 0586dae9b4c7
----> 5be9009b0d8e
Removing intermediate container 0586dae9b4c7
Step 3/5 : RUN yum install -y dnsmasq
----> Running in f01fddf6cf3b
Loaded plugins: fastestmirror, ovl
Determining fastest mirrors
 * base: mirrors.cn99.com
 * extras: mirrors.cn99.com
 * updates: mirrors.aliyun.com
Resolving Dependencies
--> Running transaction check
----> Package dnsmasq.x86_64 0:2.66-21.el7 will be installed
--> Processing Dependency: systemd-sysv for package: dnsmasq-2.66-21.el7.x86_64
--> Running transaction check
----> Package systemd-sysv.x86_64 0:219-30.el7_3.9 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package Arch Version Repository Size
=====
Installing:
dnsmasq x86_64 2.66-21.el7 base 229 k
Installing for dependencies:
systemd-sysv x86_64 219-30.el7_3.9 updates 64 k
Transaction Summary
```

```

=====
Install 1 Package (+1 Dependent package)

Total download size: 293 k
Installed size: 468 k
Is this ok [y/d/N]: Exiting on user command
Your transaction was saved, rerun it with:
  yum load-transaction /tmp/yum_save_tx.2017-07-30.07-20.YqMqm5.yumtx
The command '/bin/sh -c yum install -y dnsmasq' returned a non-zero code: 1

```

我们可以看到 `-t` 参数指定了新的镜像的仓库和名称，在这个例子中，`netdevops` 是仓库名，`dnsmasq` 是镜像名。在构建镜像的时候还可以添加一个标签，格式为“镜像名: 标签”。上面的例子中可以写成：

```
$ sudo docker build -t="netdevops/dnsmasq:v1.0.0" .
```

命令的最后还有一个“.”，表示让 Docker 读取当前目录下的 Dockerfile 文件构建镜像。这个参数是不能省略的。

镜像构建完后，我们可以通过如下命令来查询构建后的镜像情况。

```

$ sudo docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
netdevops/dnsmasq   latest       5be9009b0d8e     23 minutes ago  193MB
$ sudo docker inspect 5be9009b0d8e
<略>

```

Docker 的功能很丰富，其设计的内容和操作方式远不及上述提到的这些内容。由于本书并不是一个 Docker 相关的书籍，因此不会详细介绍 Docker 命令和参数，只是通过几个简单的小例子让大家对其有一个初步认识，以为本章的后续内容打好基础。关于 Docker 的详细内容，读者可以通过其他资料来进一步了解，也可以参考 <https://docs.docker.com> 的文档。

## 6.2 TFTP 服务器

接下来的内容都会用到 `dnsmasq` 这个软件。这个软件是 2001 年首次发布的开源软件，其提供了 TFTP、DNS、DHCP 等功能。不过，它是一款轻量级应用的软件，比较适合小规模的应用场景，通常可以提供的服务规模在 1000 并发会话左右。在中小型规模的网络管理中，这个软件还是可以胜任的。因为，其软件的资源占用率低，且易于配置。对于网络工程师，这是一款不错的入门级服务软件。

TFTP 是小型文件的传送协议，网络工程师经常会用到 TFTP 服务。比如，可以使用 TFTP 给网络设备传送操作系统进行软件升级，可以在 ZTP 的过程中为网络设备传送初始化的配置文件，还可以为 IP 电话等硬件传送固件文件等。下面将介绍如何使用 Docker 来启动一个 TFTP 的服务。

### 6.2.1 定制一个 TFTP 服务镜像

首先我们需要定制一个 TFTP 的镜像。在定制镜像之前，我们需要考虑如下几件事情：

- ❑ TFTP 使用的是 UDP 69 端口。
- ❑ TFTP 传送的文件不能包含在镜像文件中。

我们先来创建一个 Dockerfile，用于构建 TFTP 服务镜像。

```
FROM alpine:latest
MAINTAINER Xin Yu "yuxin@netdevops.cn"
RUN apk -U add dnsmasq && rm -rf /var/cache/apk/*
VOLUME /tftp
EXPOSE 69/udp
ENTRYPOINT ["dnsmasq", "-k", "--enable-tftp", "--tftp-root=/tftp"]
```

在这个 Dockerfile 中，基础 Linux 镜像没有使用 6.1.5 节中的 CentOS，而是用了 alpine Linux。因为这个 Linux 更加轻量化，整个镜像只有 5MB 左右，而一个 CentOS 大于 200MB 左右。这么小的镜像系统非常适合作为 Docker 环境的基本操作系统使用。

接下来，我们来解释一下 Dockerfile 的内容。

第 1 行的 FROM 关键字指定了 alpine 为基本镜像，并且版本为最新的版本。

第 2 行的内容在 6.1.5 节中已介绍，这里不再赘述。

第 3 行使用 APK 包管理工具安装了 dnsmasq，并且删除安装过程中的缓存文件。

第 4 行的 VOLUME 命令用于保存数据以及共享容器间的数据，这里指定的目录将会绕开 Union FS。我们在 6.1.2 节中提到过 Docker 镜像中的文件系统在创建后是只读的，如果我们希望保存一些数据，我们需要把这些文件保存到操作系统上。通过这个命令，我们可以在 Docker 镜像中指定一个目录，这个目录在 Docker 启动的时候通过参数来指定一个映射到操作系统的目录中（参考 6.2.2 节中的命令参数）。这个目录保存的内容将直接保存在操作系统上，而不是保存在 Docker 镜像中。

最后两行的内容在 6.1.5 节中已经介绍过。不过，这里对 dnsmasq 多了两个参数，从名字上可以很容易理解这两个参数的功能，这里不再展开叙述。

最后，我们需要基于 Dockerfile 开始构建镜像文件。

```
$ sudo docker build -t "netdevops/tftp" .
Sending build context to Docker daemon 2.048kB
Step 1/5 : FROM alpine:latest
--> 7328f6f8b418
Step 2/5 : RUN apk -U add dnsmasq inotify-tools && rm -rf /var/cache/apk/*
--> Using cache
--> efb38fa76170
Step 3/5 : VOLUME /tftp
--> Running in c31f4dd4dee9
--> 90fc59e7ca8c
Removing intermediate container c31f4dd4dee9
Step 4/5 : EXPOSE 69/udp
```



```

---> Running in ccf5be968de6
---> 2149fa399929
Removing intermediate container ccf5be968de6
Step 5/5 : ENTRYPOINT dnsmasq -k --enable-tftp --tftp-root=/tftp
---> Running in 7757257fe3d3
---> 923482808dfd
Removing intermediate container 7757257fe3d3
Successfully built 923482808dfd
Successfully tagged netdevops/tftp:latest

$ sudo docker images
REPOSITORY          TAG             IMAGE ID         CREATED          SIZE
netdevops/tftp      latest         923482808dfd    About a minute ago  4.38MB

```

## 6.2.2 启动一个 TFTP 服务器的容器

构建完 TFTP 镜像后，就可以启动 TFTP 容器了。这里启动的 Docker 容器比第一个启动的容器要复杂一些。

```

$ sudo docker run -d -p 69:69/udp \
--cap-add=NET_ADMIN \
-v /home/lab/dockerfile/tftp:/tftp \
netdevops/tftp

```

说明如下。

- ❑ `-d`: 使启动的容器在后台运行，通常作为一个服务而言，都是需要的。
- ❑ `-p`: 指定映射的端口，将容器的端口映射到主机的端口上。
- ❑ `--cap-add`: 添加 Linux 的能力。默认 Docker 中的 root 用户的权限还是会受到一定的限制的，如果 Docker 的服务需要用到小于 1024 的一些低位端口，我们就需要给予容器内的 root 用户更高的权限。不然，这样的端口服务将无法打开。其中，关键字 `NET_ADMIN` 就是网络部分的权限（注意需要大写）。
- ❑ `-v`: 挂载的目录，指定将主机上的目录挂载到容器内的某个目录上。这样的目录通常用于保持数据的持久性，在这个例子中，是为了让容器直接访问到主机上的目录。这个目录也是 TFTP 文件存放的地方。

启动完容器后，我们可以通过如下操作来检查和查看容器的情况。

```

$ sudo docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                    NAMES
00aa3edb0a47   netdevops/tftp "dnsmasq -k ..."     4 seconds ago Up 3 seconds
0.0.0.0:69->69/udp          modest_elion

```

## 6.2.3 服务的检查

当容器运行的时候，通常需要进入容器内查看容器的状态，可以使用如下命令：

```
$ sudo docker exec -i -t 00aa3edb0a47 /sh
```

docker 后的子命令是 exec，表示执行 Docker 中的某个命令。-i -t 参数的含义和图 6-4 中使用的参数是一样的，可以参考 6.1.5 节。其后需要指定容器的 ID，最后需要明确运行什么命令。通过这个命令，我们就可以通过交互式的方式获得一个 tty 伪终端，并且运行 sh 的 shell 环境，这样我们就进入了这个容器。在这里，我们可以监控容器的运行情况。

## 6.3 DNS 服务器

DNS (Domain Name Service, 域名服务) 是互联网的基础服务之一，我们无时无刻不在使用这个服务。这个服务的基本功能是实现 IP 地址和域名之间的转换。也就是说，我们可以通过域名来查找 IP 地址，也可以通过 IP 地址来查找域名。我们常用的是第一种方式。我们在进行网络管理的时候每天要接触很多 IP 地址，IP 地址的形式是不方便人的记忆和识别的。我们可以使用 DNS 从 IP 到域名的服务来方便我们的日常工作。

```
$ traceroute www.github.com
traceroute: Warning: www.github.com has multiple addresses; using 192.30.253.113
traceroute to github.com (192.30.253.113), 64 hops max, 52 byte packets
1  shn-xinyu3- 2.448 ms 1.976 ms 1.994 ms
<略>
15 ae-4.r25.tkokhk01.hk.bb.gin.ntt.net (129.250.4.5) 48.152 ms 48.523 ms 46.727 ms
16 ae-1.r24.osakjp02.jp.bb.gin.ntt.net (129.250.2.80) 102.007 ms 104.066 ms 109.349 ms
17 ae-4.r22.snjsca04.us.bb.gin.ntt.net (129.250.2.130) 270.739 ms 418.188 ms 417.836 ms
18 ae-7.r23.asbnva02.us.bb.gin.ntt.net (129.250.6.238) 419.110 ms 367.223 ms 423.029 ms
19 ae-2.r05.asbnva02.us.bb.gin.ntt.net (129.250.2.22) 412.471 ms 417.233 ms 417.780 ms
20 ae-0.a01.asbnva02.us.bb.gin.ntt.net (129.250.5.182) 304.157 ms
    ae-1.a01.asbnva02.us.bb.gin.ntt.net (129.250.5.188) 283.837 ms
    ae-0.a01.asbnva02.us.bb.gin.ntt.net (129.250.5.182) 301.791 ms
21 xe-0-0-25-1.a01.asbnva02.us.ce.gin.ntt.net (128.241.3.22) 424.791 ms 442.278 ms 408.041 ms
22 * * *
23 * * *
24 lb-192-30-253-113-iad.github.com (192.30.253.113) 475.749 ms 369.725 ms 281.218 ms
```

从上面的例子看，ntt 对很多设备的互联接口都做了域名解析。通过 DNS 服务，我们可以很清楚地知道流量经过了哪些设备的哪些端口。在完成本节的内容后，大家可以知道如何搭建这样的 DNS 服务器了。

### 6.3.1 构建 DNS 镜像

和 TFTP 类似，我们也需要先创建一个 Dockerfile 文件，通过这个文件来完成 DNS 镜像的构建。

```
FROM alpine:latest
MAINTAINER Xin Yu "yuxin@netdevops.cn"
RUN apk -U add dnsmasq && rm -rf /var/cache/apk/*
VOLUME /dns
```

```
EXPOSE 53 53/udp
ENTRYPOINT ["dnsmasq", "-k"]
```

这个文件和 6.2.1 节的相比，改动并不多，第 1 行也使用了 alpine Linux。第 2、3 行保持不变。第 4 行做了简单的修改，这里希望把一些配置文件保存到一个特殊的目录下。第 5 行修改了端口信息。DNS 需要用到 TCP 53 和 UDP 53 这两个端口，所以 EXPOSE 命令后为 53 53/udp。最后一行其实减少了一些启动参数。dnsmasq 默认配置文件中启用了 DNS 服务，因此在启动参数中可以不添加其他参数。

根据 6.2.1 节的步骤，我们还需要使用 docker build 来构建镜像。这次，我们把镜像名称改为了 netdevops/dns。

```
$ sudo docker build -t "netdevops/dns" .
Sending build context to Docker daemon 2.048kB
Step 1/6 : FROM alpine:latest
----> 7328f6f8b418
Step 2/6 : MAINTAINER Xin Yu "yuxin@netdevops.cn"
----> Running in 8aab0299af05
----> b782b45b38c7
Removing intermediate container 8aab0299af05
Step 3/6 : RUN apk -U add dnsmasq && rm -rf /var/cache/apk/*
----> Running in 9052da812a92
fetch http://dl-cdn.alpinelinux.org/alpine/v3.6/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.6/community/x86_64/APKINDEX.tar.gz
(1/1) Installing dnsmasq (2.76-r4)
Executing dnsmasq-2.76-r4.pre-install
Executing busybox-1.26.2-r5.trigger
OK: 4 MiB in 12 packages
----> e3548c81cb01
Removing intermediate container 9052da812a92
Step 4/6 : VOLUME /dns
----> Running in ddb6a1ae164b
----> 1289c5837b6a
Removing intermediate container ddb6a1ae164b
Step 5/6 : EXPOSE 53 53/udp
----> Running in 117582fa9d0b
----> 84b82661a207
Removing intermediate container 117582fa9d0b
Step 6/6 : ENTRYPOINT dnsmasq -k
----> Running in b393a98ea5ef
----> 7f4bc2f735c6
Removing intermediate container b393a98ea5ef
Successfully built 7f4bc2f735c6
Successfully tagged netdevops/dns:latest
```

### 6.3.2 启动和配置 DNS

我们先启动一个简单的 DNS 服务。

```
$ sudo docker run -d -p 53:53 -p 53:53/udp \
    --cap-add=NET_ADMIN netdevops/dns \
    --address /test.netdevops.cn/1.1.1.1
```



通过这个命令，我们在启动 dnsmasq 的时候添加了一个域名到 IP 的解析记录。

我们可以通过如下方式进行测试。

使用 dig 命令：

```
$ dig @localhost test.netdevops.cn

; <<>> DiG 9.10.3-P4-Ubuntu <<>> test.netdevops.cn @localhost
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 4426
;; flags: qr aa rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
test.netdevops.cn.          IN      A

;; ANSWER SECTION:
test.netdevops.cn. 0      IN      A      1.1.1.1

;; Query time: 0 msec
;; SERVER: ::1#53(::1)
;; WHEN: Fri Sep 01 10:07:33 HKT 2017
;; MSG SIZE rcvd: 51
```

使用 nslookup 命令：

```
$ nslookup test.netdevops.cn localhost
Server:          localhost
Address:         ::1#53
Name:            test.netdevops.cn
Address: 1.1.1.1
```

这里简单地解释一下上面用到的两个 DNS 查询工具。这两个工具都是非常常用的，一个是 dig，另一个是 nslookup。首先，dig(domain information groper 域信息搜索器)是一个简写，从名字上就可以知道它的功能。上面的命令所表示的含义为，通过 DNS 服务器 localhost（也就是服务器自身）来查询 test.netdevops.cn 的 IP 地址。我们可以看到在“;; ANSWER SECTION:”下面有服务器返回的结果，这个结果和我们在启动 DNS 服务器时的参数是一致的。nslookup 工具也是一个用于域名查询的工具。和 dig 工具相比，其功能相对简单一些。大家如果对这两个查询工具想有更多的了解可以参考下面这两个链接。

- ❑ <ftp://ftp.isc.org/isc/bind9/cur/9.10/doc/arm/man.dig.html>;
- ❑ <https://linux.die.net/man/1/nslookup>。

### 6.3.3 用 DNS 记录设备的接口与 IP 的对应关系

在 6.3.2 节中，我们启动了一个非常简单的 DNS 服务，其数据只有一条记录。这样的方式显然不太适合日常工作。

为了便于大家理解，我们先设定一个应用场景。这个场景在 6.3 节开始处有过简单描

述,即我们是否可以创建一个 DNS 服务来转换设备接口名称与 IP 地址的对应关系。某个小型的数据中心的网络设备链路互联表如表 6-2 所示。

表 6-2 pod1.sha.netdevops.cn

节点 A	节点 A 接口名	节点 A IP 地址	节点 Z IP 地址	节点 Z 接口名	节点 Z
Spine1	TE-0/0/0	192.168.130.1	192.168.130.2	TE-0/0/0	Leaf1
spine1	TE-0/0/1	192.168.130.5	192.168.130.6	TE-0/0/0	Leaf2
spine1	TE-0/0/2	192.168.130.9	192.168.130.10	TE-0/0/0	Leaf3
spine1	TE-0/0/3	192.168.130.13	192.168.130.14	TE-0/0/0	Leaf4
spine2	TE-0/0/0	192.168.130.17	192.168.130.18	TE-0/0/1	Leaf1
spine2	TE-0/0/1	192.168.130.21	192.168.130.22	TE-0/0/1	Leaf2
spine2	TE-0/0/2	192.168.130.25	192.168.130.26	TE-0/0/1	Leaf3
spine2	TE-0/0/3	192.168.130.29	192.168.130.30	TE-0/0/1	Leaf4

我们可以把表 6-2 中的 IP 地址与接口名称做一个对应,如表 6-3 所示。在表 6-3 中,我们可以看到接口的表示方式发生了一些变化。这是因为在 RFC1035 中对 DNS 中使用的字符进行了规定,我们可用的字符是大小写字母、数字和“-”这几个字符。虽然,在后续的 RFC 中增加了很多的字符。但是,出于兼容性的考虑还是采用最早 RFC1035 中定义的字符。另外,在 RFC1035 中对每个子域名的长度也限制为 63 个字符,总的长度不能为 255 个字符。

表 6-3 IP 与 DNS 记录对应关系

IP 地址	DNS 记录
192.168.130.1	te-0-0-0.spine1.pod1.sha.netdevops.cn
192.168.130.5	te-0-0-1.spine1.pod1.sha.netdevops.cn
192.168.130.9	te-0-0-2.spine1.pod1.sha.netdevops.cn
192.168.130.13	te-0-0-3.spine1.pod1.sha.netdevops.cn
192.168.130.17	te-0-0-0.spine2.pod1.sha.netdevops.cn
192.168.130.21	te-0-0-1.spine2.pod1.sha.netdevops.cn
192.168.130.25	te-0-0-2.spine2.pod1.sha.netdevops.cn
192.168.130.29	te-0-0-3.spine2.pod1.sha.netdevops.cn
192.168.130.2	te-0-0-0.leaf1.pod1.sha.netdevops.cn
192.168.130.6	te-0-0-0.leaf2.pod1.sha.netdevops.cn
192.168.130.10	te-0-0-0.leaf3.pod1.sha.netdevops.cn
192.168.130.14	te-0-0-0.leaf4.pod1.sha.netdevops.cn
192.168.130.18	te-0-0-1.leaf1.pod1.sha.netdevops.cn
192.168.130.22	te-0-0-1.leaf2.pod1.sha.netdevops.cn
192.168.130.26	te-0-0-1.leaf3.pod1.sha.netdevops.cn
192.168.130.30	te-0-0-1.leaf4.pod1.sha.netdevops.cn

我们把表 6-3 的内容保存为 pod1.conf 文件。文件内容如下：

```
$ cat pod1.conf
192.168.130.1    te-0-0-0.spine1.pop1.sha.netdevops.cn
192.168.130.5    te-0-0-1.spine1.pop1.sha.netdevops.cn
192.168.130.9    te-0-0-2.spine1.pop1.sha.netdevops.cn
192.168.130.13   te-0-0-3.spine1.pop1.sha.netdevops.cn
192.168.130.17   te-0-0-0.spine2.pop1.sha.netdevops.cn
192.168.130.21   te-0-0-1.spine2.pop1.sha.netdevops.cn
192.168.130.25   te-0-0-2.spine2.pop1.sha.netdevops.cn
192.168.130.29   te-0-0-3.spine2.pop1.sha.netdevops.cn
192.168.130.2    te-0-0-0.leaf1.pop1.sha.netdevops.cn
192.168.130.6    te-0-0-0.leaf2.pop1.sha.netdevops.cn
192.168.130.10   te-0-0-0.leaf3.pop1.sha.netdevops.cn
192.168.130.14   te-0-0-0.leaf4.pop1.sha.netdevops.cn
192.168.130.18   te-0-0-1.leaf1.pop1.sha.netdevops.cn
192.168.130.22   te-0-0-1.leaf2.pop1.sha.netdevops.cn
192.168.130.26   te-0-0-1.leaf3.pop1.sha.netdevops.cn
192.168.130.30   te-0-0-1.leaf4.pop1.sha.netdevops.cn
```

现在，我们先关掉原来的容器，然后重新启动新的容器。

```
$ docker ps --format "table {{.ID}} \t {{.Image}}"
CONTAINER ID      IMAGE
9ee0c744f903      netdevops/dns
```

```
$ docker stop 9ee0c744f903
```

```
$ docker run -d -p 53:53 -p 53:53/udp \
    -v /home/lab/dockerfile/dns:/dns \
    --cap-add=NET_ADMIN \
    netdevops/dns \
    --addn-hosts=/dns/pod1.conf
```

在这三个命令中，第一个命令为了显示的简洁，对输出结果做了一些过滤，保留了 CONTAINER ID 和 IMAGE 两个字段。第二个命令用于停止本小节一开始启动的容器。第三个命令则用于启动一个新的容器，这个容器包含了 pod1.conf 文件中的 IP 与域名的对应关系。

接下来我们用 dig 或者 nslookup 来测试一下这些记录。我们先测试一下 192.168.130.1 对应的域名情况。

dig 命令：

```
$ dig -x 192.168.130.1 @localhost | grep PTR
;1.130.168.192.in-addr.arpa.    IN PTR
1.130.168.192.in-addr.arpa. 0 IN PTR te-0-0-0.spine1.pop1.sha.netdevops.cn.
```

nslookup 命令：

```
$ nslookup -query=ptr 192.168.130.1 localhost
Server:      localhost
Address:     ::1#53
```



```
1.130.168.192.in-addr.arpa name = te-0-0-0.spine1.pop1.sha.netdevops.cn.
```

这里的 dig 命令和 nslookup 命令与上一次相比都有一些变化。其变化就是为了查询 PTR 记录。当我们在使用 traceroute 时，其通过查询 PTR 记录来获得 IP 地址所对应的域名。只要你所使用的 DNS 服务器有相应的 IP 地址的 PTR 记录，你在使用 traceroute 的时候就会和 6.3 节开始的例子一样实现相同功能。

除了测试单个的记录是否能正常解析之外，我们还可以利用 Linux 的工具对其进行批量测试。

```
$ cat pod1.conf | awk '{print $1}' | xargs -I {1} sh -c "nslookup -query=ptr {1}
localhost | grep name"
1.130.168.192.in-addr.arpa name = te-0-0-0.spine1.pop1.sha.netdevops.cn.
5.130.168.192.in-addr.arpa name = te-0-0-1.spine1.pop1.sha.netdevops.cn.
9.130.168.192.in-addr.arpa name = te-0-0-2.spine1.pop1.sha.netdevops.cn.
13.130.168.192.in-addr.arpa name = te-0-0-3.spine1.pop1.sha.netdevops.cn.
17.130.168.192.in-addr.arpa name = te-0-0-0.spine2.pop1.sha.netdevops.cn.
21.130.168.192.in-addr.arpa name = te-0-0-1.spine2.pop1.sha.netdevops.cn.
25.130.168.192.in-addr.arpa name = te-0-0-2.spine2.pop1.sha.netdevops.cn.
29.130.168.192.in-addr.arpa name = te-0-0-3.spine2.pop1.sha.netdevops.cn.
2.130.168.192.in-addr.arpa name = te-0-0-0.leaf1.pop1.sha.netdevops.cn.
6.130.168.192.in-addr.arpa name = te-0-0-0.leaf2.pop1.sha.netdevops.cn.
10.130.168.192.in-addr.arpa name = te-0-0-0.leaf3.pop1.sha.netdevops.cn.
14.130.168.192.in-addr.arpa name = te-0-0-0.leaf4.pop1.sha.netdevops.cn.
18.130.168.192.in-addr.arpa name = te-0-0-1.leaf1.pop1.sha.netdevops.cn.
22.130.168.192.in-addr.arpa name = te-0-0-1.leaf2.pop1.sha.netdevops.cn.
26.130.168.192.in-addr.arpa name = te-0-0-1.leaf3.pop1.sha.netdevops.cn.
30.130.168.192.in-addr.arpa name = te-0-0-1.leaf4.pop1.sha.netdevops.cn.
```

上面一系列的命令对所有的记录都进行了一次查询测试。在这个命令中，awk 和 grep 在第 5 章中已有详细的介绍，这里不再赘述。对于 xargs 的命令，大家可以参考 <https://zh.wikipedia.org/wiki/Xargs>。

到此，DNS 的基本功能已经正常。我们可以用它来为网络提供基础服务。在日常的工作中，我们经常会遇到增加、删除或修改一些 DNS 的记录。但是，每次在修改后都需要重新启动容器，这里我们可以通过下面的方式来让 dnsmasq 进程快速地重读一些数据文件（这个是 dnsmasq 的特性，而非容器的特性），比如上面例子中的 pod1.conf 文件。

```
$ sudo docker kill -s HUP <容器ID>
```

## 6.4 搭建 DHCP 服务器

DHCP (Dynamic Host Configuration Protocol, 动态主机设置协议) 同样是我们每天都会使用到的基本协议。网络工程师对这个服务并不陌生，因为很多的路由器或者交换机支持这个协议。在大部分的应用场景中，DHCP 最少会分配如下信息：

- ❑ 用户的 IP 地址；
- ❑ 用户 IP 地址的子网掩码；
- ❑ 用户 IP 地址的租约时间；
- ❑ 用户的网关地址；
- ❑ 用户的 DNS 服务器地址。

下面我们就通过 dnsmasq 的容器来实现 DHCP 的基本功能。

### 6.4.1 构建 DHCP 镜像

和 TFTP（6.2 节）与 DNS（6.3 节）一样，我们需要创建一个 Dockerfile 文件为构建 DHCP 镜像提供配置信息。在前面的 dnsmasq 中，我们都使用了默认的配置文件。但在这里，我们希望定制 dnsmasq 的配置文件。我们需要创建一个名为 dhcp.conf 的配置文件。

首先，这里给出 dhcp.conf 的配置文件的内容：

```
$ cat dhcp.conf
#listen interface is eth0
interface=eth0
dhcp-range=172.18.0.50, 172.18.0.150,255.255.255.0,24h
dhcp-option=6,8.8.8.8,8.8.4.4
dhcp-option=3,172.18.0.1
dhcp-leasefile=/dnsmasq/dnsmasq.leases
```

在这个配置文件中，dhcp-range 后面指定的是地址池的相关内容，其中 172.18.0.50 与 172.18.0.150 分别是地址池的起始地址与结束地址，255.255.255.0 是子网掩码，24h 是地址的租约时间。接下来 dhcp-option=6,8.8.8.8,8.8.4.4 指的是分配给用户的 DNS 服务器地址，DHCP 使用了 option 6 发送 DNS 服务器信息。随后的这行 dhcp-option=3 告诉用户的默认网关是什么，在 DHCP 中也叫作 router。最后一行指定的是在地址分配后，分配记录文件的位置。

其次，这里给出 Dockerfile 的内容。

```
FROM alpine:latest
MAINTAINER Xin Yu "yuxin@netdevops.com"
RUN apk -U add dnsmasq && rm -rf /var/cache/apk/*
# dhcp configure file
VOLUME /etc/dnsmasq.d/
# use to save dhcp leases informations
VOLUME /dnsmasq/
EXPOSE 67/udp
ENTRYPOINT ["dnsmasq", "-k"]
```

上面的内容和之前出现的并没有太多的差别。dnsmasq 的默认配置文件会加载 /etc/dnsmasq.d/ 目录下所有以 “.conf” 结尾的文件作为配置文件。

最后，我们需要使用 docker build 命令来构建镜像。

```
$ sudo docker build -t "netdevops/dhcp" .
Sending build context to Docker daemon 643.1kB
Step 1/6 : FROM alpine:latest
----> 7328f6f8b418
Step 2/6 : MAINTAINER Xin Yu "yuxin@netdevops.com"
----> Using cache
----> b782b45b38c7
Step 3/6 : RUN apk -U add dnsmasq && rm -rf /var/cache/apk/*
----> Using cache
----> e3548c81cb01
Step 4/6 : VOLUME /etc/dnsmasq.d/
----> Using cache
----> 7a32fa8b25c6
Step 5/6 : VOLUME /dnsmasq
----> Using cache
----> aaf3a9fbfe3b
Step 6/6 : EXPOSE 67/udp
----> Using cache
----> e68c28a4de34
Successfully built e68c28a4de34
Successfully tagged netdevops/dhcp:latest
```

## 6.4.2 启动和配置 DHCP 服务

6.4.1 节已经给出了配置文件。现在我们把这个配置文件放在 /home/lab/dockerfile/dhcp 目录下，然后通过下面的命令来启动 DHCP 服务。

```
$ sudo docker run -d \
-v /home/lab/dockerfile/dhcp:/etc/dnsmasq.d \
--cap-add=NET_ADMIN --net dhcp netdevops/dhcp
```

在这里，我们增加了一个参数 `--net dhcp`。这个参数是为了便于后面对 DHCP 的测试，在 Docker 环境中单独创建了一个网络。我们可以通过下面的命令来查看 Docker 环境中存在的网络（其中除了 DHCP 之外，其他三个网络都是系统默认的）。

```
$ sudo docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
f3b87a4dd228	bridge	bridge	local
5c287260aefa	dhcp	bridge	local
11642d929bce	host	host	local
583d3daacbe7	none	null	local

对于 DHCP 这个网络，我们可以使用下面的命令进行创建。关于如何在 Docker 中构建更加复杂的网络环境，可以参考 Docker 的手册，这里就不再展开赘述。

```
$ sudo docker network create dhcp
```

为了便于进行 DHCP 的测试，我们又创建了一个 Linux 容器，这个容器包含了 `dhtest` (<https://github.com/saravana815/dhtest>) 用于测试 DHCP 的工具。这里有 `dhtest` 容器的构建方法。其中 `dhtest.tar` 是在 CentOS 下编译好的文件。构建完这个测试容器后进行启动。



```
FROM centos:latest
MAINTAINER Xin Yu "yuxin@netdevops.com"
RUN yum install -y net-tools
ADD dhctest.tar /var/local/
```

这里给出 dhctest 的测试结果。

```
$ ./dhctest -i eth0 -V -f
Using Ethernet source addr: 02:42:ac:12:00:03
Using DHCP chaddr: 02:42:ac:12:00:03
DHCP discover sent          - Client MAC : 02:42:ac:12:00:03
DHCP offer received        - Offered IP : 172.18.0.61

DHCP offer details
-----
DHCP offered IP from server - 172.18.0.61
Option no - 53, option length - 1
    OPTION data (HEX)
        02
    OPTION data (ASCII)

DHCP server - 192.168.1.1
Lease time - 1 Days 0 Hours 0 Minutes
Option no - 58, option length - 4
    OPTION data (HEX)
        00 00 A8 C0
    OPTION data (ASCII)
        ? ?
Option no - 59, option length - 4
    OPTION data (HEX)
        00 01 27 50
    OPTION data (ASCII)
        'P
Subnet mask - 255.255.255.0
Option no - 28, option length - 4
    OPTION data (HEX)
        C0 A8 01 FF
    OPTION data (ASCII)
        ? ? ?
Router/gateway - 172.18.0.1
DNS server - 8.8.8.8
DNS server - 8.8.4.4
-----

DHCP request sent - Client MAC : 02:42:ac:12:00:03
DHCP ack received - Acquired IP: 172.18.0.61
```

## 6.5 小结

到这里，我们完成了 TFTP、DNS 以及 DHCP 这三个服务的搭建过程。通过搭建服务的过程，我们也熟悉了 Docker 的一些基本功能。通过 Docker，我们可以方便地实现服务的

创建、删除以及可伸缩等。我们在后续的章节中还会用 Docker 的方式来快速部署一些应用与开发环境。因此，笔者建议大家花一些时间来了解 Docker 的更多细节。

随着这一章的结束，我们基础篇的部分也随之结束了。在基础篇中，笔者希望通过介绍一些 Linux 相关的工具来提高大家的工作效率。本章还介绍了在 DevOps 领域比较流行的 Docker 工具，笔者希望通过这些内容的介绍让传统的网络工程师有兴趣深入 NetDevOps。

从下一章开始，我们将开始提高篇的内容。在提高篇中，我们将要开启编程的相关内容。希望通过对编程相关内容的学习，大家可以更好地提高工作的效率与准确性。



### 第三篇 *Part 3*

## 提 高 篇

从本篇开始，我们就要开始接触编程的内容了。对于 NetDevOps 而言，编程是不可避免的，而且学好它并不是一朝一夕的事。因为编程涉及的内容是相当多且复杂的，有编程语言的学习、编程思想的学习、应用框架的学习等。如何快速地学习一些编程知识和技巧并能应用到我们的日常工作中，这应该是大量没有编程基础的网络工程师所期望的。对于那些有编程基础的人而言，也许希望能利用现成的库或框架来快速开发网络相关的应用。

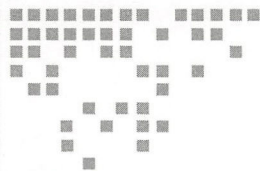
本篇涉及如下两个方面的内容。

1) 编程语言的基础内容。关于编程语言的选择，第 2 章已经叙述，大家可以参考 2.2 节的相关内容。这里我们会重点介绍两种语言：Bash 和 Python。

2) 在编程中较常见的数据类型。XML 格式和 JSON 格式的数据常用于网络设备和应用程序之间的信息交互。YAML 格式的数据具有很强的可读性，常常作为配置文件使用。另外，YANG 是一种数据结构的定义语言。我们可以通过 YANG 文件来定义前面提到的各种格式的数据结构。

.....





## Chapter 7 第 7 章

# Linux 编程基础

从本章开始，我们就要进入编程的环节了。首先，我们会了解到 Bash 编程的一些基础知识，然后，结合 Expect 工具完成一些对网络设备交互式的操作。

## 7.1 Bash 编程基础

Bash 是 Bourne Shell 的替代品，它是 GNU 的一个项目，其英文缩写来自 **Bourne-Again Shell**。而 Bourne Shell 是 1977 年左右由史蒂夫·伯恩编写的一个 shell 解释器。现在主流的 Linux 发行版都默认使用 Bash 作为其 shell 解释器。可以使用如下命令查看当前使用的 shell 解释器是什么。

下面是 Ubuntu 14.04 系统默认安装的输出结果。

```
$ env | grep SHELL
SHELL=/bin/bash
```

另外，你还可以使用如下命令来查看系统支持哪些类型的 shell。  
这里是笔者的 Mac OS X：

```
yuxin-macbookpro:~ cat /etc/shells
/bin/bash
/bin/csh
/bin/ksh
/bin/sh
/bin/tcsh
/bin/zsh #这个是笔者单独安装的shell
```

Bash 编程是一种脚本类编程。脚本类编程的一个特点是语言转换器只提供一个解释器，

而没有编译器。每次程序都依赖于解释器才能执行。其优点如下：具有很好的可移植性，只要有解释环境就可以运行程序。随之而来的缺点也正是这个解释器，运行环境必须提供一个解释器才能正常运行程序，并且其运行速度往往不如编译型语言快。不过，随着计算机的速度越来越快，有时候大家更加看重程序的可移植性，这给解释型语言带来了很大的发展空间。

## 7.2 第一个 Bash 程序

大家在学习编程的时候都会从最著名的“Hello World”程序开始。我们也遵从这个惯例从它开始。我们的第一个 Bash 程序为 hello.sh，下面是 hello.sh 的内容。

```
$ cat hello.sh
#!/bin/bash
# This is my first bash script
echo "Hello World! This is NetDevOps coder."
```

现在我们来逐行地解释一下这个程序。所有的 Bash 脚本程序都应该起始于第 1 行的“#!”，其后面的内容说明系统应该使用哪个解释器来解释后续的代码。这样的代码在很多脚本语言中都存在。比如 Python 脚本的开头也常常是“#!/usr/bin/python”。

第 2 行是注释内容。在 Bash 程序中，除了以“#!”开头的行之外，所有以“#”开头的行都是注释内容。在编写程序时，多写注释是一个好的习惯。因为更多的注释内容便于其他人看懂和理解你的代码，而很多时候这个“其他人”又往往是自己（一段时间后再来看这个代码的自己）。

第 3 行是这个程序真正运行的内容，这里只是在屏幕上输出了一行文字。对于 Bash 脚本而言，最简单的 Bash 程序就是一组 shell 命令的罗列而已。

编写完这个程序，我们就需要来运行它了。运行这个程序的方法主要有以下三种：

第一种方法是在这个程序的前面直接加上解释的程序，这里就是 Bash。在这种方法中，程序的第一行就不需要加了。因为我们已经指定了解释器。其运行结果如下。

```
$ bash hello.sh
Hello World! This is NetDevOps coder.
```

第二种方法是使用“点”命令来运行。它的表现形式有两种。

第一种形式：

```
$ . hello.sh
Hello World! This is NetDevOps coder.
```

在上面的这个命令中，“.”和运行程序之间存在一个空格。这个空格是不能省去的。“.”和程序之间没有空格就变成了另外的含义了。

第二种形式：

```
$ source hello.sh
Hello World! This is NetDevOps coder.
```

这里 source 的作用和第一种形式的 “.” 一样。

需要指出的是，如果使用这种方法的任何一种形式，那么其解释器都是当前的 shell 解释器。可以通过 7.1 节描述的方法来检查当前使用的默认 shell 解释器。

第三种方法是给脚本程序添加可执行的权限，然后通过 “./” 前缀来运行，其含义是运行当前目录下的 hello.sh 代码。

```
$/hello.sh
-bash: ./hello.sh: Permission denied
$ chmod +x hello.sh
$ ./hello.sh
Hello World! This is NetDevOps coder.
```

我们可以看到，如果不给文件添加可执行的权限，其执行后系统会报错。在这里，我们并没有指定这个文件的解释器是什么，系统会根据代码第一行的内容来选择解释器。

## 7.3 变量

### 1. 定义变量

对于一个编程语言而言，最基本的元素是变量。Bash 并没有区分变量值的类型。我们在定义变量时也非常简洁，不需要事先声明一个变量，直接赋值就可以了。例如：

```
hostname="spine_sw1"           //在赋值的时候，等号的两边不能有空格
```

Bash 语言关于变量的命名原则和大多数的编程语言比较类似，我们需要遵守如下几个原则。

- 1) 必须以字母开头，字母为 a~z 以及 A~Z，数字为非首字符。
- 2) 中间不能有空格，可以使用下划线 ( \_ )。
- 3) 不能使用标点符号和特殊字符。
- 4) 不能使用 Bash 的关键字作为变量。
- 5) 变量是区分大小写的。例如，Hostname 和 hostname 是两个不同的变量。

### 2. 使用变量

当我们定义完变量后，在使用变量的时候，需要在变量的前面加符号 “\$”，或者在变量外面使用 {} 并且加上符号 “\$”。例如：

```
echo $hostname
echo ${ip_address}/24
```

在使用变量 ip\_address 时，需要通过 “{}” 来区分其后面的字符。



### 3. 删除变量

当我们不再使用变量的时候，可以通过 `unset` 命令来删除变量。例如：

```
unset ip_address
```

这里在删除变量的时候并没有使用 “\$” 符号。如果使用了 “\$” 符号，那么 `unset` 后的值将是变量的值，而不是变量名了。

### 4. 变量的作用域

在 Bash 中，根据变量的作用域可以将变量分为局部变量和全局变量。局部变量是指其作用域只能限制在它被声明的 Bash shell 中。也就是说，每个启动的 Bash 进程都存在一个变量的空间，其作用域仅限于这个空间，这也就是所谓的局部变量。举例来说，我们可以启动两个单独的 Bash 进程，对于远程的设备，你可以登录两次设备，那么这两个单独的 Bash 进程里的变量是完全独立的，参见图 7-1。可以看到，我们在上下两个窗口中都登录了一台 Linux 服务器。这时，我们在上一个窗口中定义了一个变量 `hostname` 并赋值为 `R1`，我们可以读取到这个变量的值的。但是，在下一个窗口中，我们却找不到变量 `hostname` 的值。

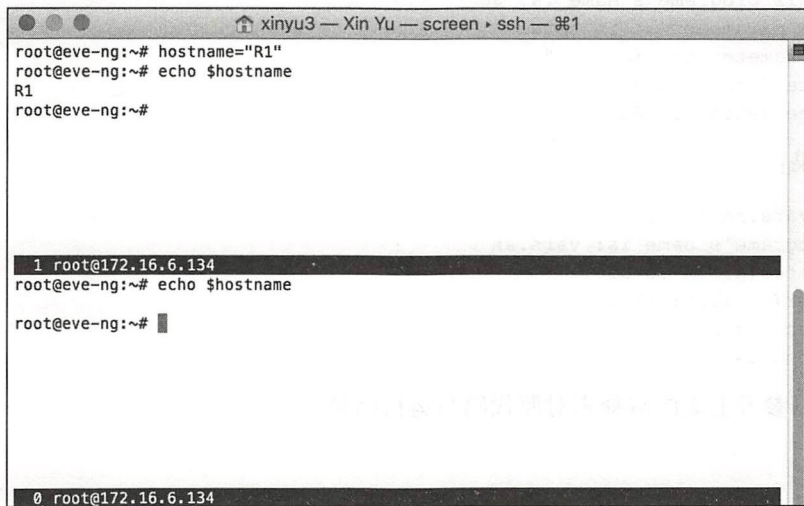


图 7-1 Bash 局部变量

全局变量有时候也称为环境变量，默认情况下，Bash 的变量是局部变量。全局变量的含义是，其子 Bash 是可以继承当前 Bash 的变量的。我们可以通过 `export` 来声明这种变量的类型。可以通过如下方式来了解全局变量。

```
$ export ip_address=10.1.1.1
$ bash
$ echo $ip_address
10.1.1.1
```

在第一行中，`export` 命令声明了一个全局的 `ip_address` 变量，然后通过 Bash 命令启动

一个子 Bash 进程。这时我们在这个子 Bash 中就可以使用 `echo $ip_address` 来访问这个变量。但是，如果变量是局部变量，那么我们在子 Bash 进程中将无法获得这个变量。

```
$ interface="ge-0/0/0"
$ bash
$ echo $interface
```

这里变量 `$interface` 的内容是空的，因为上面定义的变量不是全局变量。

## 5. 特殊变量

Bash 有一些特殊的只读变量，这些变量在程序运行的时候才能确定其值。其最为常用的就是位置变量，这些变量与程序的名字以及其参数有关。其中“`$0`”表示程序自身，“`$1`”表示程序的第一个参数，“`$2`”就是第二个参数，以此类推。如果参数个数多于 10 个，就需要使用“`${}`”来进行定义。除此以外，“`$#`”表示参数的个数，“`$@`”或者“`$*`”表示其全部参数。我们可以通过下面的代码来演示。

```
$ cat vars.sh
#!/bin/bash
echo "This programe's name is: $0"
echo "$# parameter(s) in total"
echo "Parameter(s) list: @$@"
echo "The first is $1"
echo "The second is $2"
```

运行结果：

```
$ bash vars.sh r1 r2
This programe's name is: vars.sh
2 parameter(s) in total
Parameter(s) list: r1 r2
The first is r1
The second is r2
```

大家可以参考上面的解释来对照代码与运行结果。

## 7.4 数组

7.3 节介绍了单个变量的基本内容。下面我们来认识一下数组。数组其实是一种特殊的数据结构。数组中的每一项被称为一个元素，而元素和我们 7.3 节中的变量是非常类似的。在 Bash 的数组中，并不要求每一个元素都是相同的数据类型。假设，我们现在要保存某台路由器上所有接口的 IP 地址信息。如果不用数组来表示 IP 地址信息，我们就需要对每个接口都定义一个变量。显然，这是一个比较麻烦且可行性很差的办法。

### 7.4.1 定义数组

数组的定义方法和变量的定义方法非常类似。它有如下几种定义方式。

### 1) 使用 declare 命令。

```
$ declare -a interfaces_array
$ interfaces_array[0]="1.1.1.1/30"
$ interfaces_array[1]="1.1.1.5/30"
$ interfaces_array[2]=" N/A"
```

这里先用 declare 命令来完成一个变量名的定义，其后的两行命令是向数组中添加两个元素。 Bash 数组的下标是从 0 开始的（大部分编程语言的编号都是从 0 开始的）。

与变量类似，数组是很宽松的。我们随时都可以向数组中添加元素。

### 2) 不使用 declare 命令，直接赋值给变量。

```
$ bgp_peers=("10.1.1.100" "10.1.1.101" "10.1.1.102")
```

### 3) 在定义时给特定位置元素赋值。

```
$ interfaces_status=[0]="up" [2]="up" [4]="up")
```

## 7.4.2 数组取值

刚刚了解如何定义数组和给数组的元素赋值后，接下来介绍如何从数组中获取值。

### 1) 我们可以直接使用下标来获取数组中的值。例如：

```
$ echo ${interfaces_array[0]}
1.1.1.1/30
$ echo ${interfaces_array[1]}
1.1.1.5/30
```

### 2) 我们可以使用如下两种方式来获取所有的元素。

```
$ echo ${interfaces_array[@]}
1.1.1.1/30 1.1.1.5/30 N/A
$ echo ${interfaces_array[*]}
1.1.1.1/30 1.1.1.5/30 N/A
```

从输出的结果中，我们无法看出这两种方式的差别。其具体的差别，大家在以后写代码的过程中会有所体会。

## 7.4.3 获取数组的长度

数组的长度即数组中包含了多少个元素。那些没有值的元素将不会被记录在其中。

```
$ echo ${#bgp_peers[*]}
3
$ echo ${#bgp_peers[@]}
3
$ echo ${#interfaces_status[*]}
3
$ echo ${#interfaces_status[@]}
3
```



可以使用“@”或“\*”先获得元素，然后在变量名的前面使用字符“#”获取元素的个数。

如果把“@”或“\*”替换为元素的下标，那么将获得具体相应元素的长度。

```
$ echo ${#bgp_peers[1]}
10
$ echo ${#interfaces_status[0]}
2
$ echo ${#interfaces_status[1]}
0
```

#### 7.4.4 截取数组的内容

有时我们并不需要获取数组的全部内容，而是想获取其中的一部分内容，这时我们需要截取其中的一部分内容。

假如我们想从 `bgp_peers` 中获取第二个元素和第三个元素。

```
$ echo ${bgp_peers[@]:1:2}
10.1.1.101 10.1.1.102
```

由于数组的编号是从 0 开始的，因此，第二个元素的下标是 1。然后，我们获取从这个位置开始及其后的两个元素（对应程序语句中的数字 2），这样我们就得到了第二个元素和第三个元素了。

例如，我们想获得第三个元素：

```
$ echo ${bgp_peers[@]:2:1}
10.1.1.102
```

#### 7.4.5 替换元素中的内容

将元素中的内容进行替换：

```
$ echo ${bgp_peers[@]/10.1.1/172.16.1}
172.16.1.100 172.16.1.101 172.16.1.102
```

在这个命令中，首先，`bgp_peers` 中元素的值并没有改变，只是在输出的时候进行了替换。其次，大家需要注意的是替换的格式。在上面的格式中，只有两个“/”。结束部分不需要使用“/”。

如果需要修改原来的值，那么我们需要做的是把修改后的值再一次赋给变量。

```
$ bgp_peers=(${bgp_peers[@]/10.1.1/172.16.1})
$ echo ${bgp_peers[@]}
172.16.1.100 172.16.1.101 172.16.1.102
```

#### 7.4.6 删除数组中的元素或者数组

删除数组中的元素和删除一个变量使用了相同命令：

```
$ unset bgp_peers[1]
$ echo ${bgp_peers[@]}
172.16.1.100 172.16.1.102
```

`unset` 命令除了可以用于删除变量和数组中的元素外，也可以用于删除数组。例如：

```
$ unset bgp_peers
```

## 7.5 运算符

在 Bash 中，运算符主要包含算术运算符、位运算符、自增 / 自减运算符、比较运算符、字符串运算符、文件操作运算符、逻辑运算符等。下面我们主要介绍算术运算符、位运算符、自增 / 自减运算符。

### 7.5.1 算术运算符

算术运算符、位运算符、自增 / 自减运算符主要用于整数计算，因为 Bash 只支持整数计算。所有涉及小数的数都将被舍去小数部分（不是四舍五入，而且全部舍去），而保留整数部分数值。对于数值的计算，有以下几种方式：

1) 使用内置的 `let` 命名。其形式如下：

```
let "变量名=表达式"。例如：
$ let "a=1+2"
$ echo $a
3
```

如果不使用 `let` 命令，那么等号后面的值会被认为是一个字符串，而不是数值的计算。

```
$ a=1+2    #这里等号"="的两边不能有空格
$ echo $a
1+2
```

2) 使用 `$[]` 或 `$(())` 来做运算。后面的小括号是两个小括号（这里并没有书写错误）。这种方式比较适用于一些简单的计算。例如：

```
$ echo $[2*3]
6
$ echo $((2**3)) # 2的三次方
8
$ echo $(2**3)  #如果少了一个括号，那么里面的表达式就会被认为是一个命令
2**3: command not found
```

另外，还有一些其他运算。

1) 除法。例如：

```
$ echo $[3/4]
0
```

这个结果应该等于 0.75，但是在 Bash 中却等于 0。正如前面所说的，Bash 会删除小数

部分而只保留整数部分。因此，这个值就变成了 0。

2) 余数运算。例如：

```
$ echo ${10%3}
1
```

3) 使用 expr 来做运算。

在使用 expr 命令的时候需要注意的是它对后面的表达式有不同的要求：数字和操作符之间使用空格隔开。如果不隔开，则其会被认为是一个字符串。并且，expr 前不需要使用 echo 这个打印字符的命令。例如：

```
$ expr 2-1          #表达式没有空格，因此不会做数值计算。只给出字符串内容
2-1
$ expr 2 - 1        #这里的表达式中的数值和表达式之间是有空格的
1
```

4) 使用 declare 命令。在 Bash 中，declare 是一个内置的命令。这个命令可以用于声明变量的类型。默认情况下，Bash 中的变量类型都是字符串类型。因此，当需要进行数值计算时，我们需要一个数值变量。使用 declare -i 就可以定义变量为一个整数变量。

```
$ x=2+3
$ echo $x
2+3
```

我们之前见过类似的例子。现在我们先来声明 x 变量为一个整数变量。

```
$ declare -i x
$ x=2+3
$ echo $x
5
```

在实际工作学习中，大家可以根据具体情况来使用上面的四种方法。另外，在 Bash 中，如果你确实需要进行非整数的计算，你可以使用 bc 这个工具。这个工具是一个 GNU 项目，提供了一个高精度的计算结果。但是，这个工具并不是默认安装在 Linux 的发行版中的，我们需要手动进行安装。bc 工具的参考文档见 <https://www.gnu.org/software/bc>，这里我们就不给出更多的介绍了。

### 7.5.2 位运算符

位运算的基本操作主要有六种，包括左移动、右移动、位与、位或、位异或以及位非。这几种运算方法是很常用的处理二进制数的方法，因此在处理 IP 地址信息时是非常有用的。我们先简单介绍一下这几种操作。在介绍之前，我们先看图 7-2，图 7-2 中给出了 IP 地址 192.168.2.1 对应的二进制表示方式。这个对应关系，对于网络工程师来说应该是不会陌生的。

192	168	2	1
11000000	10101000	00000010	00000001

图 7-2 IP 地址的二进制表示方式



## 1. 左移动

如果我们需要把 IP 地址转换为十进制表示方式，可以使用如下命令来完成。

```
$ let "ip1=${192<<24} + ${168<<16} + ${2<<8} + ${1}"
$ echo $ip1
3232236033
```

在上面的命令中，我们使用了左移动运算。左移动后，后面的位会用 0 来填充。

数字 192 的二进制表示方式为 11000000。其左移动 24 位后，相当于在后面添加了 24 个 0（二进制），即 11000000 00000000 00000000 00000000。同理，168 左移动 16 位为 00000000 10101000 00000000 00000000（在最高的 8 个位中用 0 进行填充）。2 左移动 8 位为 00000000 00000000 00000010 00000000。

1 表示为 00000000 00000000 00000000 00000001。因此，这四个数相加就可以得到 11000000 10101000 00000010 00000001。这个二进制数用十进制可表示为 3232236033。

因此，通过左移动运算可以很快地把一个 IP 地址转换为十进制表示方式。获得十进制表示方式后，我们就可以对 IP 地址进行加减处理了。

## 2. 右移动

右移动是左移动的逆操作，我们可以通过右移动来处理十进制到 IP 地址的转换。在上一个例子中，我们得到了一个十进制数 3232236033。我们可以通过如下方式来获得 IP 地址。

为了便于理解，我们还是把 3232236033 表示为二进制形式：11000000 10101000 00000010 00000001。

首先，我们获得 IP 地址的第一个八位值。

```
$ echo ${3232236033>>24}
192
```

其次，我们要获得第二个八位值。我们先要减去第一个八位值，然后将得到的值右移 16 位。

我们可以做如下操作：

```
$ echo ${3232236033-3232236033>>24<<24}>>16}
168
```

经过 3232236033>>24<<24 操作，我们得到的值（二进制表示方式）为 11000000 00000000 00000000 00000000。经过 3232236033-3232236033>>24<<24 操作，我们得到的值（二进制表示方法，首 8 位补 0）为 00000000 10101000 00000010 00000001。

然后，我们将这个数右移 16 位就可以得到所需值（二进制表示方法），即 00000000 10101000 00000010 00000001。

同理可以获得第三个八位和第四个八位的值。

第三位：

```
$ echo ${3232236033-3232236033>>16<<16}>>8]
2
```

第四位:

```
$ echo ${3232236033-3232236033>>8<<8]
1
```

这样，我们就获得了一个 IP 地址的十进制表示方式了。

综合上面的几个步骤:

```
$ ip=3232236033
$ echo "${$ip>>24}.${($ip-$ip>>24<<24)>>16}.${($ip-$ip>>16<<16)>>8}.${$ip-$ip>>8<<8}"
192.168.2.1
```

3. 位与运算

位与运算符为“&”。位与运算的运算规则见表 7-1。第一个数(十进制为 168)和第二个数(十进制为 240)都写成了二进制的形式。在相同位置，如果两个数的值都为 1，那么结果也是 1，其他情况都为 0。

表 7-1 位与运算的运算规则

第一个数	1	0	1	0	1	0	0	0
第二个数	1	1	1	1	0	0	0	0
结果	1	0	1	0	0	0	0	0

```
$ echo ${168 & 240}
160
```

将位与运算与右移动运算结合起来可以更加直观地获得由数字到 IP 地址的转换。其操作如下:

```
$ ip=3232236033
$ echo "${$ip>>24&255}.${$ip>>16&255}.${$ip>>8&255}.${$ip&255}"
192.168.2.1
```

我们还是把这个过程用二进制来表示。3232236033 为 11000000 10101000 00000010 00000001。

获得第一个八位值:

```
IP地址:11000000 10101000 00000010 00000001      #右移动24位
掩码:   11111111
结果:   11000000      #即十进制192
```

获得第二个八位值:

```
IP地址:11000000 10101000 00000010 00000001      #右移动16位
掩码:   00000000 11111111      #在前面补了8个0
结果:   00000000 10101000      #即十进制168
```

获得第三个八位值:

```
IP地址:11000000 10101000 00000010 00000001 #右移动16位
掩码: 00000000 00000000 11111111 #在前面补了16个0
结果: 00000000 00000000 00000010 #即十进制2
```

获得第四个八位值:

```
IP地址:11000000 10101000 00000010 00000001 #右移动16位
掩码: 00000000 00000000 00000000 11111111 #在前面补了24个0
结果: 00000000 00000000 00000000 00000001 #即十进制2
```

#### 4. 位或运算

位或运算符为“|”。其运算规则如下: 如果对应的位置有 1, 那么其结果就为 1。位或运算可以用于获取 IP 地址的广播地址。其计算步骤如下:

- 1) 对子网掩码取反, 即 1 为 0, 0 为 1。
- 2) IP 地址与取反后的掩码进行位或计算, 得到的结果就是该网段的广播地址。

例如, 计算 192.168.10.2 255.255.255.0 的广播地址。255.255.255.0 取反的结果为 0.0.0.255。这就是我们网络工程师熟悉的反掩码。

```
$ echo "$[192|0].$[168|0].$[10|0].$[2|255]"
192.168.10.255
```

我们可以再计算一个非类地址“172.16.10.2 255.255.240.0”。255.255.240.0 取反后的反掩码为 0.0.15.255。

```
$ echo "$[172|0].$[16|0].$[10|15].$[2|255]"
172.16.15.255
```

#### 5. 位异或运算

位异或运算符为“^”。其运算规则如下: 如果对应的位置值相同, 其结果为 0, 不同则为 1。其可以将子网掩码转换为反掩码。例如: 子网掩码是 255.255.248.0, 我们可以将其和 255.255.255.255 做位异或运算。

```
$ echo "$[255^255].$[255^255].$[248^255].$[0^255]"
0.0.7.255
$ echo "$[255^255].$[255^255].$[255^255].$[252^255]"
0.0.0.3
```

思考一下如何把反掩码转换为子网掩码 (提示一下, 可以用相同的操作), 你可以在 Bash 中试一试。

#### 6. 位非运算

位非运算运算符是“~”。其运算规则如下: ~a 的值为 -(a+1), 也称加一取反。关于位非运算, 我们了解一下即可, 使用不是很多。



### 7.5.3 自增 / 自减运算

自增运算符为“++”，自减运算符为“--”。我们通过几个例子来说明其运算情况。

```
$ a1=10
$ a2=20
$ echo ${++a1}
11
$ echo ${++a1}
12
$ echo ${--a2}
19
```

这个运算符只能和变量一起使用，其不能和表达式一起使用。

## 7.6 测试

在写程序时，我们经常需要根据具体的情况做出一些判断。在 Bash 中，较常见的判断有三大类，分别是对文件的判断、对字符串的判断和对整数的判断。除此以外，还需要对逻辑进行操作。下面我们先分别对这几种判断进行描述。

### 7.6.1 测试语法的结构

测试语法的结构有两种。一种是使用 test 命令对表达式进行测试。格式如下：

```
test 表达式
```

另一种方法是使用“[]”。需要注意的是，在“[]”和表达式之间需要有空格。格式如下：

```
[ 表达式 ]
```

在测试完表达式后，我们可以读取“\$?”的系统默认变量来获得上一次测试的结果。例如：

```
$ ls /etc/hosts
/etc/hosts
$ echo $?
0
```

当文件不存在的时候，返回的值为非 0 值。

### 7.6.2 文件测试

在刚才的例子中已经测试了文件是否存在，不过测试表达式并不是这么做的。我们在测试文件是否存在的时候需要使用 -e 操作来完成。

例如，用 test 测试：

```
$ test -e /etc/hosts
$ echo $?
0
$ test -e /etc/host
$ echo $?
1
```

再如，用 [] 测试：

```
$ [ -e /etc/hosts ]
$ echo $?
0
```

表 7-2 给出了一些常用的文件测试参数。

表 7-2 文件测试参数

文件测试参数	描 述
-e	当文件或目录存在时返回真，否则返回假
-d	当目录存在时返回真，否则返回假
-f	当文件存在时返回真，否则返回假
-x	当文件为可执行文件时返回真，否则返回假
-r	当文件可读时返回真，否则返回假
-w	当文件可写时返回真，否则返回假
-s	当文件存在且文件大小大于 0 时返回真，否则返回假
file1 -nt file2	当 file1 比 file2 文件新时返回真，否则返回假
file1 -ot file2	当 file1 比 file2 文件旧时返回真，否则返回假

基于表 7-2 中的参数，下面给出一个测试文件的程序。

```
$ cat file_test.sh
#!/bin/bash
read -p "Please input file name: " filename

if [ ! -f "$filename" ]; then
    echo "The file does not exist!"
    exit 1
fi

if [ -r "$filename" ]; then
    echo "$filename is readable!"
fi

if [ -x "$filename" ]; then
    echo "$filename is executable!"
fi
```

```
if [ -s "$filename" ]; then
    echo "$filename is exist and size > 0."
fi
```

运行结果如下：

```
$ ./file_test.sh
Please input file name: file_test.sh
file_test.sh is readable!
file_test.sh is executable!
file_test.sh is exist and size > 0.
```

这里涉及的一些语法在后面会具体提到。

7.6.3 整数测试

整数测试是比较简单的测试。其测试参数见表 7-3。

表 7-3 整数测试参数

整数测试参数	描 述
-eq	等于，为 equal 缩写
-gt	大于，为 great than 缩写
-lt	小于，为 less than 缩写
-ge	大于等于，为 great equal 缩写
-le	小于等于，为 less equal 缩写
-ne	不等于，为 not equal 缩写

例如：

```
bash$ [ 2 -eq 2 ]
bash$ echo $?
0
bash$ [ 2 -gt 3 ]
bash$ echo $?
1
bash$ [ 2 -lt 3 ]
bash$ echo $?
0
```

7.6.4 字符串测试

字符串测试有等于、不等于、大于、小于以及是否为空的测试。其测试参数见表 7-4。

表 7-4 字符串测试参数

字符串测试参数	描 述
-z	当字符串为空时返回真，否则返回假



(续)

字符串测试参数	描 述
-n	当字符串非空时返回真，否则返回假
“字符串 1” = “字符串 2”	当两个字符串相同时返回真，否则返回假
“字符串 1” != “字符串 2”	当两个字符串不相同返回真，否则返回假
“字符串 1” > “字符串 2”	两个字符串按照字母排序，字符串 1 排在前面则返回真，否则返回假
“字符串 1” < “字符串 2”	两个字符串按照字母排序，字符串 1 排在后面则返回真，否则返回假

例如：

```
//先定义三个字符串变量
bash$ s1=" "
bash$ s2=" r1"
bash$ s3=" R1"

bash$ if [ -z "$s1" ]; then echo "True"; else echo "False"; fi
True

bash$ if [ -z "$s2" ]; then echo "True"; else echo "False"; fi
False

# s2 和s3大小写不一样。所以，两者不相等。
bash$ if [ "$s2" = "$s3" ]; then echo "True"; else echo "False"; fi
False

# 在ASCII字母表中，由于小写字母的值大于大写字母的值，因此r1 > R1，这里返回的值是真
bash$ if [ "$s2" \> "$s3" ]; then echo "True"; else echo "False"; fi
True
```

需要注意的是，这里的“>”和“<”需要转义。如果不使用转义符，则需要使用“[[ ]]”。例如：

```
bash$ if [[ "$s2" > "$s3" ]]; then echo "True"; else echo "False"; fi
True
```

7.6.5 逻辑关系

存在多个测试的情况下，有三种逻辑关系，其分别是逻辑非、逻辑与和逻辑或。其逻辑符号如表 7-5 所示。

表 7-5 逻辑符号

逻辑符号	描 述
!	结果取反
&&	也可使用“-a”，如果前后两个表达式都为真，其结果为真
	也可以使用“-o”，如果其中一个表达式为真，其结果为真

例如：

```
bash$ [ ! -f /etc/hosts ]
bash$ echo $?
1

bash$ [ -f /etc/hosts -a -r /etc/hosts ]
bash$ echo $?
0

bash$ [ -f /etc/hosts ] && [ -x /etc/hosts ]
bash$ echo $?
1
```

## 7.7 判断结构

程序正是有了逻辑判断才变得“智能”。Bash 主要在两个地方存在判断，一个是选择判断，另一个是循环的条件。这里说的判断是选择判断，其关键词有 if 和 case。

### 7.7.1 if 结构

其基本语法如下：

```
if 表达式; then
    命令
elif 表达式; then
    命令
fi
```

下面是一个判断带宽的例子：

```
bash$ cat bandwidth.sh
#!/bin/bash
read -p "Please input interface bandwidth(Kbps): " bandwidth

if [ $bandwidth -eq 1000000 ]; then
    echo "GigabitEthernet"
elif [ $bandwidth -eq 10000000 ]; then
    echo "TenGigabitEthernet"
elif [ $bandwidth -eq 25000000 ]; then
    echo "25GigabitEthernet"
elif [ $bandwidth -eq 4000000000 ]; then
    echo "40GigabitEthernet"
elif [ $bandwidth -eq 1000000000 ]; then
    echo "100GigabitEthernet"
fi
```

运行结果如下：

```
bash$ ./bandwidth.sh
```

```
Please input interface bandwidth(Kbps): 10000000
TenGigabitEthernet
```

## 7.7.2 case 结构

case 结构的语法如下：

```
case 变量 in
  值1) 命令;;
  值2) 命令;;
  值3) 命令;;
  *) 命令;;
esac
```

7.7.1 节中用 if 结构写的例子，如果采用 case 结构，可以写成如下形式：

```
bash$ cat band.sh
#!/bin/bash
read -p "Please input interface bandwidth(Kbps): " bandwidth

case $bandwidth in
1000000) echo "GigabitEthernet" ;;
10000000) echo "TenGigabitEthernet" ;;
25000000) echo "25GigabitEthernet" ;;
40000000) echo "40GigabitEthernet" ;;
100000000) echo "100GigabitEthernet" ;;
esac
```

运行结果如下：

```
bash$ ./band.sh
Please input interface bandwidth(Kbps): 1000000
GigabitEthernet
```

## 7.8 循环结构

与人相比，机器的一个强项是能快速地重复做一件可重复的事情。在编程的时候，我们经常会用到一些重复执行的命令。循环结构就能很好地完成这个任务，Bash 支持的循环结构有 for、while、until 以及 select。

### 7.8.1 for 结构

在 Bash 的循环结构中，for 结构是最为常见的结构。从语法结构而言，for 结构又可以分为两种结构形式，一种是带列表的 for 结构，另一种是类 C 语言的 for 结构。

#### 1. 带列表的 for 结构

带列表的 for 结构通常用于执行有限次数的循环，其循环执行的次数就是列表中元素的个数。其语法如下：



```

for 变量 in (列表)
do
    命令或语句
done

```

在下面的例子中，我们将生成一段设备的配置。其目的是在 RR（路由反射器）上建立一些 BGP 的 peer。

代码如下：

```

bash$ cat bgp.sh
#!/bin/bash

bgp_peers="10.1.1.100 10.1.1.101 10.1.1.102 10.1.1.103"
echo "router bgp 100"
for peer in ${bgp_peers}
do
    echo "neighbor $peer remote-as 100"
    echo "neighbor $peer update-source lo0"
done
echo "exit"

```

运行结果如下：

```

bash$ ./bgp.sh
router bgp 100
neighbor 10.1.1.100 remote-as 100
neighbor 10.1.1.100 update-source lo0
neighbor 10.1.1.101 remote-as 100
neighbor 10.1.1.101 update-source lo0
neighbor 10.1.1.102 remote-as 100
neighbor 10.1.1.102 update-source lo0
neighbor 10.1.1.103 remote-as 100
neighbor 10.1.1.103 update-source lo0
exit

```

这个例子首先定义了一个数组，数组包含了四个元素。这四个元素包含了四个 IP 地址。这些地址信息会在 for 语句中被遍历。紧接着一行只是输出了“router bgp 100”这个命令。由于这个语句并没有包含在 for 结构中，因此，我们可以看到其只输出了一次。接下来就是 for 结构。for 会从数组中依次获得每一个 IP 地址，然后赋值给变量 peer 并执行输出相关完整配置。最后，在 for 结构 done 的外面，输出了“exit”字符串。由于这个操作在 done 的外面，因此，它也只被输出一次。这段代码的逻辑还是很容易理解的。通过这个方法，我们就可以快速生成一段设备的配置。

## 2. 类 C 语言的 for 结构

了解 C 语言的读者一定熟悉 (i=1;i<=5;i++) 这样的形式，这个结构可以放在 Bash 的 for 结构中。其语法如下：

```
for ((表达式1;表达式2;表达式3))
do
    语句或命令
done
```

在这个结构中，表达式 1 通常是一个变量的初始化过程，在这里，变量会被赋予初始值。表达式 2 为一个判别式，当这个判别式为假时，程序将跳出循环。如果这个判别式永远为真，那么这个 for 语句将永远执行下去，即无限循环。表达式 3 通常定义为变量的变化情况，将在每次执行完一次循环后被执行（变化）一次。

上面的例子可以改写为：

```
bash$ cat bgp1.sh
#!/bin/bash

echo "router bgp 100"
for ((i=0; i<4; i++))
do
    echo "neighbor 10.1.1.10$i remote-as 100"
    echo "neighbor 10.1.1.10$i update-source lo0"
done
echo "exit"
```

运行结果如下：

```
bash$ ./bgp1.sh
router bgp 100
neighbor 10.1.1.100 remote-as 100
neighbor 10.1.1.100 update-source lo0
neighbor 10.1.1.101 remote-as 100
neighbor 10.1.1.101 update-source lo0
neighbor 10.1.1.102 remote-as 100
neighbor 10.1.1.102 update-source lo0
neighbor 10.1.1.103 remote-as 100
neighbor 10.1.1.103 update-source lo0
exit
```

我们可以看到，以上两个例子实现了一样的运行结果。

## 7.8.2 while 结构

相对 for 结构，while 结构的语法更加简洁，其语法结构如下：

```
while 表达式
do
    语句或命令
done
```

首先程序会测试表达式的值，如果表达式的值是真则进入循环，如果是假则跳出循环。每次循环执行结束后都将检查表达式中的值。这种循环叫作“前测试循环”。for 结构实现

的循环也是前测试循环。

下面我们将使用 `while` 结构来逐行地读取一个文件中的信息。

假设文件的内容如下：

```
bash$ cat routers_infos.txt
router1  10.1.1.1  23 admin admin
router2  10.1.1.2  23 admin admin
router3  10.1.1.3  23 admin admin
```

在这个文件中，第一列是设备的主机名，第二列是设备的 IP 地址信息，第三列是设备的端口信息，第四列和第五列是用户名和密码。我们使用 `while` 结构来输出这些内容。

```
bash$ cat routers_read.sh
#!/bin/bash
cat routers_infos.txt | while read LINE
do
    line=( $LINE )
    echo "hostname is ${line[0]}, IP address is ${line[1]}"
    echo "username is ${line[3]}, password is ${line[4]}"
done
```

运行结果如下：

```
bash$ ./routers_read.sh
hostname is router1, IP address is 10.1.1.1
username is admin, password is admin
hostname is router2, IP address is 10.1.1.2
username is admin, password is admin
hostname is router3, IP address is 10.1.1.3
username is admin, password is admin
```

### 7.8.3 until 结构

与 `while` 结构类似，`until` 结构也采用了运行前测试的方式。其和 `while` 不同的是，当测试结果是假时才会进入循环，一旦测试结果为真将会结束循环。其语法如下：

```
until 表达式
do
    语句或命令
done
```

`until` 结构和 `while` 结构非常类似，这里就不再展开叙述了。

### 7.8.4 select 结构

从语法上看，`select` 结构和带列表的 `for` 结构非常类似。其语法如下：

```
select 菜单 in (列表)
do
    语句或命令
done
```



虽然两者在结构上很类似，但是，`select` 结构的程序在运行的时候会把所有的元素自动生成一个从 1 开始的列表。这个列表会等待用户进行输入。因此，`select` 结构也常常和 `case` 一起来实现一个文字界面的菜单。例如：

```
bash$ cat select1.sh
#!/bin/bash
select ROUTER in R1 R2 R3 R4
do
    case $ROUTER in
        R1) echo "Hostname is R1, IP address is 10.1.1.1" ;;
        R2) echo "Hostname is R2, IP address is 10.2.2.2" ;;
        R3) echo "Hostname is R3, IP address is 10.3.3.3" ;;
        R4) echo "Hostname is R3, IP address is 10.4.4.4" ;;
        *) echo "exit" && break ;;
    esac
    echo "please select "
done
```

运行结果如下：

```
./select1.sh
1) R1
2) R2
3) R3
4) R4
#? 1
Hostname is R1, IP address is 10.1.1.1
please select
#? 2
Hostname is R2, IP address is 10.2.2.2
please select
#? 3
Hostname is R3, IP address is 10.3.3.3
please select
#? 4
Hostname is R3, IP address is 10.4.4.4
please select
#? 5
exit
```

通过这个结构，我们很容易实现一个菜单的功能。

## 7.9 函数

函数可以认为是一组命令或者语句的自定义集合。使用函数最大的好处是可以减少代码的重复编写，同时提高了代码的可读性。在 `Bash` 中，函数的定义方法如下：

```
function <函数名>() {
```

```

命令1
命令2
.....
}

```

通常情况下，函数都会有返回值。返回值是函数与其他代码之间沟通的重要信息。现在我们用函数来编写一个程序，代码如下：

```

bash$ cat func.sh
#!/bin/bash
function checkIP() {
    # 创建一个地址信息库，地址信息为10.1.1.1~10.1.1.255。这里采用数组形式存储地址
    declare -a ips
    for ((i=1; i<256; i++))
    do
        ips[$i]="10.1.1.$i"
    done
    # 检查地址是否包含在地址信息库中
    for ip in ${ips[@]}
    do
        if [ "$1" = "$ip" ]; then
            # 如果匹配到地址，则函数返回0
            return 0
        fi
    done
    # 如果遍历了所有的地址信息都没有找到，则函数返回1
    return 1
}
# 通过函数checkIP对地址进行检查
checkIP $1

# 判断函数的返回值，如果是0则说明找到了，如果是非0 则说明没有找到
if [ $? -eq 0 ]; then
    echo "$1 is found"
else
    echo "$1 can't find"
fi

```

运行结果如下：

```

bash$ ./func.sh 10.1.1.2
10.1.1.2 is found
bash$ ./func.sh 10.1.2.1
10.1.2.1 can't find

```

在这个例子中，需要匹配的地址池是临时生成的。大家可以通过 7.8.2 节中的例子来读取一个文本作为需要匹配的地址池。另外，在上面的这个例子中，IP 地址都是不带子网掩码的主机路由地址。但是在现实情况中，大部分 IP 地址是非 32 位的主机路由地址。如何比较这样的路由地址，大家可以参考 7.5 节的内容来解决这个问题。位运算是非常快的运算

操作，大家可以不用担心它的执行效率问题。如何把上面的这个例子优化得更好，这是留给大家的一个思考题。

总之，上面的例子只给出了 Bash 函数非常简单的应用。上述代码的所有内容在本章都有涉及。大家可以参考前面的内容，对这段代码进行详细的解读。更多关于 Bash 函数的内容可以参考：<http://tldp.org/LDP/abs/html/functions.html>。

## 7.10 用 expect 实现与设备的交互式操作

通过对前面内容的学习，我们已经具备了 Bash 编程的基础。为了能更加熟练地掌握和应用 Bash 基础知识，下面我们将学习使用 expect 对网络设备进行交互式的自动化操作。

### 7.10.1 expect 简介

expect 是一款基于 UNIX 系统的用于自动化控制和测试的软件工具，可以应用在很多交互式的其他软件工具中，如 Telnet、Ftp、SSH 等。这个工具会在 UNIX 使用伪终端的方式来对交互式程序进行自动化控制。其官方网站为 <http://expect.sourceforge.net>。现在提供的软件版本为 5.45。

```
Bash$ expect -v
expect version 5.45
```

expect 基本命令有四个，如表 7-6 所示。

表 7-6 expect 基本命令

命 令	作 用
send	用于向子进程中的伪终端发送字符串
expect	从子进程中的伪终端接受字符串
spawn	启动一个新的子进程
interact	允许用户进入交互模式

expect 在进行自动化控制时，模拟的是人的操作。绝大部分通过伪终端（pty）方式登录后人能进行的操作都可以借助 expect 来实现自动化控制（操作）。对于网络设备而言，特别是那些传统的老型号设备（这些设备大多没有 API，它们能提供的交互方式就是命令行），可以通过 expect 快速地实现自动化控制。当然，通过这种方式来管理设备的效率不一定高，但是通过 expect 至少可以实现人能完成的绝大部分功能，而且是以一种自动化的方式来完成。

expect 这个工具并不是系统默认安装的，需要我们手动安装。下面给出 CentOS 和 Ubuntu 的安装命令。



CentOS:

```
bash$ yum install -y expect
```

Ubuntu:

```
bash$ apt-get install -y expect
```

## 7.10.2 用 expect 实现与设备的交互

我们先用 expect 登录到一台网络设备上。其代码如下:

```
bash$ cat get_version.exp
#!/usr/bin/expect -f
set host [lindex $argv 0]
set port [lindex $argv 1]
set timeout 10
spawn telnet $host $port
send "\r"
expect "Username*"
send "admin\r"
expect "Password*"
send "admin\r"
expect "RP/0/0/CPU0*"
send "terminal length 0\r"
expect "RP/0/0/CPU0*"
send "show platform\r"
expect "RP/0/0/CPU0*"
send "exit\r"
sleep 1
```

说明如下。

第 1 行, “#!/usr/bin/expect -f” 命令和 Bash 的代码一样, 通过用来指定 expect 解释器所在的位置。默认情况下, expect 安装的路径在 /usr/bin/ 下。其中 -f 表示从文件读取命令, 仅用于使用 #! 时。

第 2 行和第 3 行, 表示从命令行读取参数。其中, \$argv 是参数数组, 使用 [lindex \$argv n] 获取, \$argv 0 为第一个参数, \$argv 1 为第二个参数。

第 4 行, 修改 expect 中的超时时间。

第 5 行, 使用 spawn 命令创建一个子进程。

第 6 行, 向子进程发送一个 “\r” 符号。有的时候, 连接成功后, 网络设备不会默认提供登录信息, 需要客户端向网络设备发送一个 “\r” 才行。

第 7 行, 使用 expect 命令等待设备的回显。这里的 expect 是 expect 工具的一个命令, 而不是 expect 应用本身, 这一点很容易被混淆。这里希望看到设备给出一个 “Username\*” 字符串。这里 “Username\*” 是一个正则表达式。当然它是区分大小写的。这个 expect 命令是一个有阻的命令。其含义是代码执行到这里时, 需要等待设备给出回应。但是, 有时

候由于网络的原因或者是设备响应慢的原因，等待回应可能需要很长时间。那么程序执行到这里会暂停，后续的任务也无法继续执行下去，这个过程称为阻塞，这也是所有的交互式操作的一个弊端。在交互式操作中，每次上传一个命令给到远端网络设备后，都需要等待远端网络设备返回回应（应答）之后才能继续往下执行，这就大大降低了程序的执行效率。特别是在修改网络设备的配置时，传统网络设备的配置存在大量的上下文关联性，这就导致程序的执行效率大大降低了。因此，对于此类操作网络设备的程序，影响其执行效率的往往不是代码的优化程度或编程语言本身的执行效率，而是这些有阻操作。

第 8 行，向设备发送“admin\r”字符串。admin 是登录网络设备需要的用户名。

第 9 行，继续等待网络设备给一个密码的提示符。这里期望的字符串是“Password\*”。

第 10 行，向设备发送登录密码。这里的密码是“admin\r”。把登录设备的用户名和密码都写在程序里面是一件非常不好的事情。如何提高登录的安全性，可以参考 3.2 节关于设备管理的部分。

第 11 行，成功登录设备后，网络设备会给一个操作的提示符。这台设备是 Cisco IOS vXR，其提示符是“RP/0/0/CPU0:<主机名>#”，因此，这里期待的字符是“RP/0/0/CPU0\*”。

第 12 行，这里向设备发送了“terminal length 0”。网络工程师对此应该不陌生。这里设置这个伪终端的长度为 0，也就是说在后续的命令中不再分屏显示内容，而是一次性全部显示完全。

第 13 行，这里和第 11 行的含义一样。

第 14 行，向设备发送“show platform\r”命令。这个命令是网络设备支持的命令，其含义这里就不做解释了。

第 15 行，和第 11 与第 13 行是相同的意思。这里也是在等待第 14 行的命令执行完成。

第 16 行，向设备发送“exit\r”命令，表示需要退出网络设备。这也是一种温和退出的方式。

第 17 行，让程序等待 1 秒钟再退出。这里也是为了能够温和地退出网络设备。如果发送完 exit 就直接结束程序，那么，也许会由于系统繁忙，这个发送给网络的字符并没有真的被发送出去，而程序已经执行完成，导致网络设备并没有真正地收到第 16 行发送的退出信息。如果每次登录设备都不是正常退出，在网络设备上也许会保留很多以前登录的会话，从而导致设备管理资源的浪费。

现在我们来运行一下这个代码。

```
# ./get_version.exp 10.1.1.1 23
spawn telnet 10.1.1.1 23
Trying 10.1.1.1...

Connected to 10.1.1.1.
Escape character is '^'.
```

```
User Access Verification
```

```
Username:
```

```
Username: admin
```

```
Password:
```

```
RP/0/0/CPU0:ios#terminal length 0
```

```
Mon Sep 11 23:02:09.681 UTC
```

```
RP/0/0/CPU0:ios#show platform
```

```
Mon Sep 11 23:02:09.761 UTC
```

Node	Type	PLIM	State	Config State
0/0/CPU0	RP(Active)	N/A	IOS XR RUN	PWR,NSHUT,MON

```
RP/0/0/CPU0:ios#
```

我们可以看到上面的代码很好地执行了预期的命令。这段代码的逻辑还是比较清晰且简单的。整个过程就是我们日常操作网络设备的情况。不过这个代码并没有涉及任何的错误处理。例如，某种情况网络设备连接不上，导致 Telnet 失败。又如，用户名或密码错误无法登录。再如，登录设备后没有权限执行 show platform 命令。这些异常的结果都是无法正确拿到设备的相关信息。对于由上面原因导致的异常，后期我们可以在代码中给出相关明确的错误提示。

### 7.10.3 用 expect 实现批量备份设备配置

上面一个例子只完成了一台设备的登录。下面我们可以结合循环结构来实现多台设备的登录，并完成设备配置的自动备份功能。具体脚本如下：

```
$ cat telnet_cmd.exp
#!/usr/bin/expect -f
set host [lindex $argv 0]
set cmd [lindex $argv 1]
set timeout 10
spawn telnet $host
send "\r"
expect "Username*"
send "admin\r"
expect "Password*"
send "admin\r"
expect "RP/0/0/CPU0*"
send "terminal length 0\r"
expect "RP/0/0/CPU0*"
send "$cmd\r"
expect "RP/0/0/CPU0*"
send "exit\r"
sleep 1
```

这部分代码和 7.10.2 节的基本一致，只修改了两行代码。其中，第 3 行修改为 set cmd [lindex \$argv 1]。这个代码的含义在 7.10.2 节已经解释了。通过这个代码，我们可以对一台



设备运行任意一个命令。我们把网络设备的 IP 地址以及需要运行的命令作为两个参数。现在，我们先单独运行这个代码。其运行结果如下：

```
$ ./telnet_cmd.exp 10.1.1.1 "show running-config"
spawn telnet 10.1.1.1

Trying 10.1.1.1...
Connected to 10.1.1.1.
Escape character is '^]'.

User Access Verification

Username:
Username: admin
Password:

RP/0/0/CPU0:ios#terminal length 0
Fri Sep 15 06:27:13.052 UTC
RP/0/0/CPU0:ios#show running-config
Fri Sep 15 06:27:13.132 UTC
Building configuration...
!! IOS XR Configuration 6.0.1
!! Last configuration change at Wed Sep 13 10:29:16 2017 by admin
!
telnet vrf default ipv4 server max-servers 10
interface MgmtEth0/0/CPU0/0
  ipv4 address 10.1.1.1 255.255.255.0
!
interface GigabitEthernet0/0/0/0
  shutdown
!
interface GigabitEthernet0/0/0/1
  shutdown
!
interface GigabitEthernet0/0/0/2
  shutdown
!
end
```

接下来我们结合 7.8.2 节中 while 结构的例子，首先把需要备份的网络设备的 IP 地址放在一个文本文件中，然后通过一个 Bash 脚本来调用上面的 expect 代码。代码如下：

```
$ cat hosts
10.1.1.1
10.1.1.2

$ cat backup.sh
#!/bin/bash
HOSTS=$1
cat $HOSTS | while read HOST
```

```
do
echo "start to backup ${HOST}'s configuration"
./telnet_cmd.exp $HOST "show running-config" > $HOST.cnf
echo "$HOST: backup finished"
done
```

现在,我们再来读一次 backup.sh 的代码。在第 2 行中,我们把需要读取的网络设备 IP 信息文件作为了 backup.sh 的第一个参数。在第 6 行中,我们调用了 telnet\_cmd.exp。在这里,我们把命令的输出内容保存到了一个文件中。其他行代码,我们就不再重复解释了。

现在我们来运行一下这个代码。

```
$ ./backup.sh hosts
start to backup 10.1.1.1's configuration
10.1.1.1: backup finished
start to backup 10.1.1.2's configuration
10.1.1.2: backup finished
$ ls -l *.cnf
-rw-r--r-- 1 root root 2010 Sep 11 14:39 10.1.1.1.cnf
-rw-r--r-- 1 root root 1819 Sep 11 14:39 10.1.1.2.cnf
```

现在我们可以看到 backup.sh 成功地备份了两台设备的配置文件,并且每台设备的信息都单独保存到了一个文件中。

这是一个简单的通过 Bash 脚本和 expect 工具一起来实现批量设备的配置保存功能的代码。当然,这个代码的功能还不是很完善,如不同设备类型的登录方式存在差异、用户名和密码都明文保存在代码中、保存的文件也没有按照日期进行保存等。但是,它毕竟实现了最简单的自动登录设备并运行一个命令的功能。在后续的章节中,我们还会不断地完善这个代码。

## 7.11 网络设备上的 Bash

现在大量网络设备的操作系统是基于 Linux 或者 FreeBSD 的。其中部分设备还提供了 Bash 的操作环境,如 Arista EOS、Cisco NX-OS 以及 Juniper JUNOS。这里我们以 Cisco NX-OS 为例子。

```
switch# run guestshell
[admin@guestshell ~]$
[admin@guestshell ~]$ env | grep -i shell=
SHELL=/bin/bash

[admin@guestshell ~]$ bash --version
GNU bash, version 4.2.46(1)-release (x86_64-redhat-linux-gnu)
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

我们可以看到 NX-OS 提供了 Bash,在 Bash 中,我们可以直接操作设备。例如:

```
[admin@guestshell ~]$ dohost "show run interface e1/1"
!Command: show running-config interface Ethernet1/1
!Time: Sat Sep 16 07:29:23 2017
version 7.0(3)I5(2)
interface Ethernet1/1
no switchport
ip address 10.1.1.1/24
no shutdown
```

```
[admin@guestshell ~]$ dohost "conf t ; interface e1/1 ; shutdown"
Enter configuration commands, one per line. End with CNTL/Z.
```

```
[admin@guestshell ~]$ dohost "show run interface e1/1"
!Command: show running-config interface Ethernet1/1
!Time: Sat Sep 16 07:31:02 2017
version 7.0(3)I5(2)
interface Ethernet1/1
no switchport
ip address 10.1.1.1/24
```

我们在 7.8.1 节中用 for 结构生成了一段 BGP 的配置。如果在 NX-OS 上，我们对这段代码稍微做一些改动，就可以为设备添加一段 BGP 的配置。代码如下：

```
[admin@guestshell ~]$ cat bgp.sh
#!/bin/bash
bgp_peers="10.1.1.100 10.1.1.101 10.1.1.102 10.1.1.103"

dohost "conf t ; feature bgp"
for peer in ${bgp_peers}
do
    dohost "conf t ; router bgp 100 ;? neighbor $peer remote-as 100 ; update-
        source loopback 0"
done
```

运行结果如下：

```
[admin@guestshell ~]$ ./bgp.sh
Enter configuration commands, one per line. End with CNTL/Z.
Enter configuration commands, one per line. End with CNTL/Z.
Enter configuration commands, one per line. End with CNTL/Z.
Enter configuration commands, one per line. End with CNTL/Z.
Enter configuration commands, one per line. End with CNTL/Z.
```

```
[admin@guestshell ~]$ dohost "show running-config bgp"
!Command: show running-config bgp
!Time: Sat Sep 16 09:49:13 2017
version 7.0(3)I5(2)
feature bgp
router bgp 100
    neighbor 10.1.1.100
        remote-as 100
```



```
update-source loopback0
neighbor 10.1.1.101
remote-as 100
update-source loopback0
neighbor 10.1.1.102
remote-as 100
update-source loopback0
neighbor 10.1.1.103
remote-as 100
update-source loopback0
```

关于 NX-OS 中 Bash 编程的更多例子可以参考 <https://developer.cisco.com/dpslate/build/site/nx-os/docs/guides/developer-guide>。

## 7.12 小结

在本章，我们学习了 Linux 的 Bash 编程基础知识。本章的内容无法涵盖 Bash 编程的所有内容，但是至少让大家对 Bash 编程有了一个初步的认识。在实际编程中，很多时候并不是功能难以实现，而是否有比较完善的异常处理机制和异常信息提供，这是编程中的一个难点，也是大家需要花更多的时间来完善的内容。

Bash 编程在系统管理中的应用是非常广泛的。在 Linux 下还有很多工具可以使用，特别是一些文本处理工具。我们在第 5 章中对较常用的工具做过一些介绍。大家可以结合本章的编程基础知识来实现更加丰富的功能。

在第 8 章，我们将要开始 Python 编程之旅，Python 语言的可读性会更好，其提供的库文件也更加丰富。如果大家对本章的内容觉得有点晦涩且不好理解，那也没有关系，因为 Python 几乎可以替代 Bash 的所有功能。

## Python 编程基础

我们从这一章开始接触和学习 Python 的内容。我们在 2.2 节中已经阐述过为什么选择使用 Python 语言，这里不再赘述。Python 编程的内容相对较多，而且都较为实用，因此第 8 章到第 12 章主要讲述的都是 Python 相关的内容。第 8 章为 Python 编程的基础部分。本章介绍 Python 的运行环境、基本数据类型、基本架构函数、Python 的标准模块和第三方模块如何使用。

### 8.1 Python 简介

Python 是一个纯粹的面向对象语言。Python 中一切皆对象，函数、模块、数字、字符串等都是对象，并且其完全支持继承、重载、派生、多重继承等面向对象语言的特点，但本书并不涉及这些面向对象的高级应用，有兴趣的读者可以在完成 Python 基础内容后自行提高学习。

#### 8.1.1 Python 的版本差异

目前，Python 有两个版本体系：一个是 Python 2，这个版本是在 2000 年 10 月发布的；另一个是 Python 3，这个版本是在 2008 年 12 月发布的。这两个版本存在一定的兼容性问题。在这里并不是想向大家介绍这两个版本具体有哪些细节差异，而是希望告诉读者，在运行 Python 代码的时候要注意 Python 版本的问题。

对于 Python 2 而言，目前（2017 年）的软件版本为 2.7.13。根据 PEP 373 的描述，Python 2.7 将于 2020 年停止更新，并且也不会有 Python 2.8 版本。这里还有一个网站专门给 Python 2 做了一个倒计时——<https://pythonclock.org>。PEP 是 Python Enhancement Pro-

posals 的缩写，其所包含的文档是指导 Python 软件的发展方向，它的作用和 IEEE 组织发布的 RFC 非常类似。现在大量的项目在逐步转向 Python 3，但是，由于存在大量的老项目，了解 Python 2 还是很有必要的。我们在选择 Python 版本的时候，应尽量使用 Python 3 版本，当其他条件有限制时也可以使用 Python 2 版本，比如目前大量网络设备上的 Python 运行环境还是 Python 2。

### 8.1.2 主机与网络设备上的 Python

通常我们会在 Linux 上运行 Python 的程序。现在较为常见的 Linux 发行版通常会默认安装 Python 解释器，其默认版本有 Python 2、Python 3。比如 CentOS 7 的默认 Python 版本就是 2.7，而 Ubuntu 16.04 的默认版本已经升级到了 3.5。无论 Linux 系统是否默认安装 Python 或者安装了什么版本的 Python，我们都可以根据自己的需要选择安装。

除了主机的操作系统之外，很多网络设备也集成了 Python 解释器，这和 7.11 节提到网络设备上有 Bash 类似。比如，Arista EOS 系统、Cisco NX-OS 系统以及 Juniper JUNOS 系统都已经集成了 Python（当然，不只限于这些系统）。我们先看看 Cisco NX-OS：

```
switch# show version | in NXOS
NXOS: version 7.0(3)I5(2)
NXOS image file is: bootflash:///nxos.7.0.3.I5.2.bin
NXOS compile time: 2/16/2017 8:00:00 [02/16/2017 17:03:27]
switch# python
Python 2.7.5 (default, Nov 5 2016, 04:39:52)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

可以看到在 NX-OS 中是 Python 2 版本。并且，NX-OS 中已经安装了一些扩展模块（关于模块的内容，后续内容会讲解到）。我可以通过下面的命令来检查已经安装好了的扩展模块。

```
switch# run bash
bash-4.2$ pip --version
pip 1.3.1 from /usr/lib64/python2.7/site-packages/pip-1.3.1-py2.7.egg (python 2.7)
bash-4.2$ pip list
argparse (1.2.1)
contextlib2 (0.4.0)
iniparse (0.3.2)
nsenter (0.1.6)
pathlib (1.0.1)
pycurl (7.19.0)
smart (1.4.1)
urlgrabber (3.9.1)
wsgiref (0.1.2)
yum-metadata-parser (1.1.4)
```



接下来看看 JUNOS 的 Python 版本情况：

```
root@junos> show version | match Junos:
Junos: 17.1R2.7

root@junos> start shell user root
root@junos:~ #
root@junos:~ # python
/usr/bin/python: Operation not permitted.
```

JUNOS 并不允许直接运行 Python 和 Python 的代码，需要通过 op 脚本来运行。下面我们写一段用于查看 Python 软件版本的 Python 代码，并运行它。

脚本需要放置的目录：

```
root@junos:/var/db/scripts/op # pwd
/var/db/scripts/op
```

ver.py 脚本内容：

```
root@junos:/var/db/scripts/op # cat ver.py
#!/usr/bin/python
import sys
print(sys.version)
```

JUNOS 运行 op 脚本的配置方式：

```
root@junos# set system scripts op file ver.py
```

在 JUNOS 上运行 Python 脚本：

```
root@junos# run op ver.py
2.7.8 (default, Jun 17 2017, 05:48:24)
[GCC 4.2.1 (for JUNOS)]
```

JUNOS 也已经安装了部分扩展模块。这些扩展模块的功能很多，如用于处理 SSH、YANG、XML 及配置模板等。

```
root@junos:/opt/lib/python2.7/site-packages # pwd
/opt/lib/python2.7/site-packages
root@junos:/opt/lib/python2.7/site-packages # ls
Crypto          google          lxml            paho            scp
concurrent      grpc            markupsafe      paramiko        thrift
ecdsa           jinja2          ncclient        pkg_resources   yaml
enum            jnpr            netaddr         pyang
```

我们可以看到这两种类型的设备已经提供了 Python 语言的支持能力，而且集成了一些常用的扩展模块。这些扩展模块中有一部分内容我们会在后续的章节中详细介绍。

通过上面的内容，我们可以清晰地了解到，Python 程序既可以运行在网络设备上，也可以运行在管理网络设备的服务器主机上，这为我们后续编写程序提供了更加灵活的方式。

### 8.1.3 构建 Python 运行环境

我们在开始编写 Python 程序之前需要有一个 Python 的运行环境和编程环境。如果程序在网络设备上运行的，那么首先需要一台网络设备。但是，一台物理的网络设备通常并不是很方便获取和随时使用，因此我们可以使用网络设备的虚拟化软件。目前，很多厂家都会有虚拟化软件，如 Arista vEOS、Cisco NX-OSv、Cisco IOS-XRv、Juniper vMX、Juniper vSRX 等。

除了网络设备，我们通常还需要在 Linux 环境中创建一个 Python 的运行与编程环境。虽然我们可以在安装了 Python 解释器的 Linux 上直接运行 Python 代码，但是随着项目的增加，不同项目的 Python 运行环境也许会有不同的要求，比如 Python 解释器版本不一样（Python 2 或 Python 3），又或者代码运行所依赖的模块不一样。创建一个虚拟的 Python 运行环境便于我们编写和运行代码。下面我们以 Ubuntu 16.04 和 CentOS 7 为例。下面构建 Python 运行环境的几种方法在 Windows、MAC OS X 都是适用的，但是会存在一些少许的差异。

#### 1. 用 virtualenv 创建 Python 虚拟运行环境

在编写 Python 程序时，通常会用到一些第三方模块（有时也称为“库”）文件，并且不同项目通常会调用不同的第三方库。如果所有项目都混在一起，这显然不方便管理。virtualenv 是一个虚拟环境管理工具，我们可以通过 pip 工具安装它。

```
$ sudo pip install virtualenv
```

安装完 virtualenv 后，我们就可以用这个工具来创建一个 Python 的虚拟环境了。

```
$ virtualenv project1
Using base prefix '/usr'
New python executable in /home/lab/project1/bin/python3
Also creating executable in /home/lab/project1/bin/python
Installing setuptools, pip, wheel...done.
```

virtualenv 会创建一个只包含标准库的 Python 运行环境，并且会在当前目录下创建一个目录，目录的名称就是 virtualenv 的参数，这里是 project1。新创建的运行环境需要的文件都被保存在这个目录下。基于这个环境，需要使用 source 命令来进入这个运行环境。

```
$ source ./project1/bin/activate
(project1) lab@ubuntu:~$
```

运行完后，在提示符前多了“(project1)”，这表示我们已经在这个新的运行环境中了。后续这个环境需要的第三方库都可以在这个环境下单独安装，而安装的文件是独立于操作系统的。

退出当前环境的命令是 deactivate。

```
(project1) lab@ubuntu:~$ deactivate
lab@ubuntu:~$
```

在相同的操作系统之间，可以通过复制这个文件夹来实现运行环境的迁移。但在实际情况中，往往会遇到一些问题，毕竟要求两个操作系统环境完全相同还存在很多的限制。

## 2. 使用 venv 创建 Python 虚拟运行环境

Python 3.3 及以上版本的标准库中有一个 `venv` 模块，这个模块实现了和 `virtualenv` 类似的功能。如果你的系统中没有 `venv` 模块，我们可以通过命令来安装 `venv` 模块。

```
lab@ubuntu:~$ sudo apt-get install python3-venv
```

创建 Python 虚拟环境的命令如下：

```
lab@ubuntu:~$ python3 -m venv venv1
```

执行完命令后，会创建一个 `venv1` 的目录。启动这个虚拟环境使用 `source` 命令进行激活。

```
lab@ubuntu:~$ source venv1/bin/activate
(venv1) lab@ubuntu:~$
```

后续内容和 `virtualenv` 方法是完全一样的。

## 3. 使用 pyenv 来创建不同 Python 版本的共存运行环境

前面两种方法的主要功能是在一个操作系统里实现不同的 Python 运行环境，其 Python 版本是相同的。如果我们希望快速创建不同 Python 版本的运行环境，`pyenv` 是一个不错的选择。`pyenv` 是一个开源的项目，它的源代码托管在 GitHub 中，GitHub 的地址为 <https://github.com/pyenv/pyenv>。

我们需要执行如下命令，先安装 `pyenv` 的依赖包。

### (1) Ubuntu 系统

```
$ sudo apt-get update
$ sudo apt-get install make build-essential libssl-dev zlib1g-dev
$ sudo apt-get install libbz2-dev libreadline-dev libsqlite3-dev wget curl
$ sudo apt-get install llvm libncurses5-dev libncursesw5-dev
```

### (2) CentOS 系统

```
$ yum install readline readline-devel readline-static
$ sudo yum install openssl openssl-devel openssl-static
$ sudo yum install sqlite-devel
$ sudo yum install bzip2-devel bzip2-libs
```

### (3) Mac OS X 系统，需要先安装 xcode

```
$ xcode-select -install
```

安装完上面的内容，我们就可以开始安装 `pyenv` 了。

```
$ curl -L https://raw.githubusercontent.com/pyenv/pyenv-installer/master/bin/pyenv-
installer | bash
```



```
$ pyenv update
```

安装完成后，在当前用户下会创建“.pyenv”的目录，所有文件都会存放在这个目录下。卸载 pyenv 软件只需要删除这个目录就可以了。

对于 Mac OS X 系统，可以通过 brew 安装 pyenv。关于 brew 的相关内容，读者可以参考 [https://brew.sh/index\\_zh-cn.html](https://brew.sh/index_zh-cn.html)，这里不再赘述。

```
$ brew install pyenv
```

查看 pyenv 可以安装的 Python 版本：

```
$ pyenv install --list
```

Available versions:

2.1.1.3

<略>

2.7.12

2.7.13

<略>

3.5.3

<略>

3.6.2

<略>

这里包含的 Python 版本有 310 多个。我们可以根据需求进行单独的安装。

```
$ pyenv install 3.6.2
```

安装一个新的 Python 版本通常需要花费一段时间。由于操作系统的环境不同，新的 Python 版本是通过源代码直接安装的。安装完成后，我们可以查看当前系统的软件版本。

```
$ pyenv versions
```

```
* system (set by /Users/xinyu3/.pyenv/version)
```

```
3.6.2
```

设置全局的 Python 版本：

```
$ pyenv global 3.6.2
```

```
$ pyenv versions
```

```
system
```

```
* 3.6.2 (set by /Users/xinyu3/.pyenv/version)
```

当我们再次执行 Python 程序的时候，使用的就是 3.6.2 版本了。

#### 4. 使用 Docker 来创建 Python 运行环境

上面几种方法解决了不同项目之间模块管理的问题，还解决了多版本共存的问题。但是，其可移植性并不好，它们对操作系统是强依赖的。我们在第 6 章中介绍过 Docker 的使用方法，使用 Docker 来构建 Python 运行环境是一个不错的选择。使用 Docker 作为运行环境不仅可以用在 Linux 系统上，甚至在一个网络设备中也是可以实现。比如 Arista EOS

系统就可以支持 Docker, Cisco、Juniper 等厂家的 OS 也在逐步支持 Docker 的运行环境。hub.docker.com 提供了大量的 Python 容器, 我们可以直接使用或自己定制。

```
$ docker search python
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
python	Python is an interpreted, interactive, obj...	2128		[OK]
django	Django is a free web application framework...	579		[OK]
pypy	PyPy is a fast, compliant alternative impl...	108		[OK]
kaggle/python	Docker image for Python scripts run on Kaggle	70		[OK]
amazon/aws-eb-python	AWS Elastic Beanstalk Python Image	15		

<略>

以上就是几种常用的 Python 环境的构建方法, 我们可以根据不同的使用需求进行选择。

### 8.1.4 缩进在 Python 中的重要性

缩进是构成 Python 语法的重要组成部分。相同层次的语句必须具有相同的缩进。在 PEP8 中, 建议使用四个空格作为缩进, 而不建议使用制表符 (Tab 键), 更不建议使用制表符和空格混用的方式。

由于 Python 使用缩进作为语言块的分层, 这大量的新手写出来的程序都具有不错的可读性。一个程序除了其执行结果的正确性很重要之外, 代码的可读性也是非常重要的。笔者在学习 Python 之前使用过 C 和 Perl 等语言, 起初对 Python 的缩进也非常不习惯, 但随着时间的推移越来越喜欢这样的形式; 当使用了一段时间后, 回头再看看之前写过的代码, 对强制缩进可以说是无比喜欢。因为有了强制缩进后, 读自己之前没有太多注释的代码也不会觉得非常吃力。

如果你第一次接触 Python 语言, 那么请拿出你的耐心认真地做好代码缩进, 这样可以养成很好的编程习惯, 并且会在后期受益匪浅。

## 8.2 基本数据类型

在 Python 语言中, 变量并没有严格的类型要求, 但是数据的类型是有区分的。比如, “Router1” 和 “100” 就是不同的数据类型, 前者是字符串, 而后者是整数。不同的数据类型有不同的处理方式。本节将介绍 Python 的基本数据类型以及它们的基本操作。

在这个部分, 我们会使用 Python 的交互模式进行举例。只要在 Bash 中输入 “python” 或者 “python3” 就可以进入 Python 的交互模式。

```
lab@ubuntu:~$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

## 8.2.1 数字

### 1. 简单计算

数字是最简单的数据类型。在 Python 交互式解释器中，我们可以把它当作一个功能强大的计算器。

```
>>> 5*7
35
>>> 78162349*8102370
633300271667130
>>> (79803+72548)*(98743-5049)
14274374594
```

前面几个计算好像都没有什么问题。但是下面这个计算好像出错了（在 Python 2 中）。

```
>>> 1/2
0
```

这是因为，Python 2 默认两个整数相除时得到的结果会取整数。但是，如果有一个数是小数，就不会出现这个问题。例如：

```
>>> 1.0/2
0.5
>>> 1/2.0
0.5
>>> 1.0/2.0
0.5
```

虽然，这样可以解决这个问题，但是在实际编写程序中并不是很方便。我们如果希望 Python 2 的除法执行的操作是我们普通数学的除法，我们需要在 Python 2 解释器中执行如下命令：

```
>>> from __future__ import division
>>> 1/2
0.5
```

这个特征也是 python 2 和 python 3 的一个区别。

Python 还可以处理很大的数字，例如：

```
>>> 123456789123456789123456789**2
15241578780673678546105778281054720515622620750190521L
>>> 123456789123456789123456789**2+10
15241578780673678546105778281054720515622620750190531L
```

我们可以看到，上面是一个 38 位的数字的二次方再加上一个数字，Python 能进行精确的计算。这一点在很多的编程语言中是不容易做到的。

### 2. 不同进制的数值转换

1) 二进制到十进制的转换，例如：



```
>>> 0b1000
8
```

字母 b 前为 0 (数字零)。输入二进制数时, Python 解释器会自己将其转化为十进制数。

2) 十进制到二进制的转换, 例如:

```
>>> bin(32)
'0b100000'
```

这里用到了 bin 方法。

3) 十六进制到十进制的转换, 例如:

```
>>> 0xFF
255
```

4) 十进制到十六进制的转换, 例如:

```
>>> hex(128)
'0x80'
```

## 8.2.2 列表

Python 内置了几种序列, 列表、元组以及字符串是常见的三种形式。其中, 列表是可以被修改的, 而元组和字符串是不能被修改的。这里的不能修改指的是一旦被创建后就不能修改其内容和顺序。关于其不可修改性, 我们在 8.2.3 节再做具体说明。

创建一个列表的方法为, 列表中的每个元素通过逗号分开, 最外面使用方括号 “[]”。一个空的列表可以用一对方括号来表示。例如:

```
>>> devices = []
>>> host = ["R1", "1.1.1.1", 22]
```

host 列表包含了三个元素, 前两个是字符串, 最后一个数字。在 Python 的列表中, 每个元素的数据类型是可以不同的。Python 列表能容纳的元素个数是无限大的, 其容量大小受限于硬件。

### 1. 索引与切片

列表的元素是通过其下标来访问的。列表的下标从 0 开始编号。在上面 host 列表中, 如果我们希望获取第一个元素, 我们可以使用 host[0]。例如:

```
>>> host[0]
'R1'
```

如果我们要获取最后一个元素, 我们可以使用以下两种方式。

```
>>> host[2]
22
>>> host[-1]
22
```

这里下标 2 很好理解，因为编号从 0 开始，第三个元素的下标就是 2 了。如果下标编号是负数，那么就意味着是从列表的最后开始数，-1 就是最后一个，-2 就是倒数第二个。使用负数下标是一种很有用的形式，因为我们有时候并不知道列表的长度是多少（这是由于在定义列表时没有指定其长度）。例如：

```
>>> host[-2]
'1.1.1.1'
```

如果我们希望获得后面两个元素，可以使用如下命令：

```
>>> host[1:3]
['1.1.1.1', 22]
>>> host[-2:]
['1.1.1.1', 22]
```

这种获取一个列表中多个元素的方法通常被称为切片。在第一个例子中，第一个下标 1 表示起始位置且其值是包括的，第二个下标 3 表示结束位置，但其值是不包括的。上面这个 host 列表是没有下标为 3 的元素的，但是使用 3 代表了最后一个元素。当后面的这个值大于等于列表的长度时，则表示最后一个元素。

在第二个表达式中，下标 -2 表示倒数第二个元素的下标。“:”后没有写数字，表示到列表的末尾。如果“:”前面没有数字，则表示从第一个元素开始。

现在我们在 host 列表中添加几个元素：

```
>>> host.append("R2")
>>> host.append("2.2.2.2")
>>> host.append(22)
>>> host
['R1', '1.1.1.1', 22, 'R2', '2.2.2.2', 22]
```

列表有 append 方法，其用于在列表的尾部添加一个元素。

现在我们希望获取列表中所有 IP 地址的信息：

```
>>> host[1::3]
['1.1.1.1', '2.2.2.2']
```

这里第一个下标 1 表示列表中的元素 1，即第二个元素。“::”后的 3 表示步长为 3，即下一个取的值为第一个下标 1+3 的位置，直到列表的结束。如果这个步长为负数，则从后往前取值。我们可以参考下面这个例子。

```
>>> host[-2::-3]
['2.2.2.2', '1.1.1.1']
```

请读者观察一下这个例子和上一个例子之间的区别。在这个例子中，我们也取出了所有的 IP 地址信息，但是，其得到的列表与上一个例子不一样，其取出的列表正好和上一个例子的顺序相反。

我们可以使用负数步长这个特性来颠倒一个列表的顺序。例如：

```
>>> host[::-1]
[22, '2.2.2.2', 'R2', 22, '1.1.1.1', 'R1']
```

这个例子就实现了对一个列表取反的操作。

## 2. 操作列表

### (1) 添加元素。

前面介绍了使用 `append` 方法在列表最后添加一个元素。如果我们需要在列表的最前面加入一个元素，则可以使用 `insert` 方法。例如：

```
>>> host.insert(0, "info")
>>> host
['info', 'R1', '1.1.1.1', 22, 'R2', '2.2.2.2', 22]
```

`insert` 方法的第一参数指定需要在哪个位置插入元素，第二个参数是插入元素的内容。在列表最前面插入一个元素并不是非常推荐的方法，特别是当列表很大的时候，其时间消耗是会很大的。

### (2) 删除元素

1) 删除列表末尾的元素，使用的方法是 `pop` 方法。`pop` 方法还有一个返回值，就是被删除元素的内容。例如：

```
>>> host.pop()
22
>>> host
['info', 'R1', '1.1.1.1', 22, 'R2', '2.2.2.2']
```

2) 删除指定的元素。使用的方法是 `remove` 方法。例如：

```
>>> host.remove("info")
>>> host
['R1', '1.1.1.1', 22, 'R2', '2.2.2.2']
```

如果列表中存在相同的元素，那么将删除第一个元素。

### (3) 修改指定位置的元素

我们可以直接通过下标的方式来修改元素。例如：

```
>>> host[0]="R11"
>>> host
['R11', '1.1.1.1', 'R2', '2.2.2.2', 22]
```

## 3. 列表加法

两个列表是可以相加的，执行列表加法后，系统会把两个列表的内容合并到一起。例如：

```
>>> device = ["R3", "3.3.3.3", 22]
>>> host + device
['R11', '1.1.1.1', 'R2', '2.2.2.2', 22, 'R3', '3.3.3.3', 22]
```

通过加法运算可以把几个列表的内容相加，而原来各个列表中的值保持不变。



如果我们需要在列表 1 中加入列表 2 的内容, 可以使用 `extend` 方法。通过这个方法, 列表 1 的内容将会发生改变。例如:

```
>>> host.extend(device)
>>> host
['R11', '1.1.1.1', 'R2', '2.2.2.2', 22, 'R3', '3.3.3.3', 22]
```

#### 4. 列表乘法

列表除了支持加法外, 还支持乘法。对列表执行乘法, 可以让列表的内容复制多份。例如:

```
>>> device
['R3', '3.3.3.3', 22]
>>> device * 3
['R3', '3.3.3.3', 22, 'R3', '3.3.3.3', 22, 'R3', '3.3.3.3', 22]
```

### 8.2.3 元组

#### 1. 元组的基本定义

元组是一个序列, 其定义方式是使用圆括号“()”。例如:

```
>>> login=("R1","admin","admin@123")
>>> login
('R1', 'admin', 'admin@123')
```

这里我们定义了一个元组, 其中包含 3 个元素。我们可以使用和列表一样的索引与切片来访问其中的元素。例如:

```
>>> login[0]
'R1'
>>> login[1:]
('admin', 'admin@123')
```

#### 2. 元组的不可修改性

当我们尝试去修改元组中每个元素的值时, 我们会发现, 其值是不能被修改的。例如:

```
>>> login[0] = "R2"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

但是, 我们可以重新为变量 `login` 赋值。这是为什么呢?

```
>>> login = ("R2", "admin", "admin@123")
```

因为变量名 `login` 只是指向元组的一个名称而已 (其实它是一个内存的指针)。当我们重新给变量赋值的时候, 其实用了另外一个元组代替了原来的元组。变量的内容发生了改变, 而元组没有发生改变。我们可以通过 `id` 方法来查看变量在内存中的位置。

```
>>> login=("R1","admin","admin@123")
```

```
>>> id(login)
140376471913968      # 这是一个动态的值，每次的值并不一样
>>> login=("R2","admin","admin@123")
>>> id(login)
140376471914128      # 这是一个动态的值，每次的值并不一样
```

我们再来看看列表的情况。列表是可以被修改的，但修改后其内存的位置并没有发生变化。

```
>>> host=["R1", "1.1.1.1", 22]
>>> id(host)
140376471925392      # 这是一个动态值，每次的值并不一样。
>>> host[0]="R11"
>>> id(host)
140376471925392
```

也许你会觉得，元组没有列表那么灵活和方便，定义完值后又不能被修改，这样的元组似乎没有什么意义，我是不是可以用列表来代替元组呢？这个问题的答案，我们将在 8.2.5 节给出。

## 8.2.4 字符串

顾名思义，字符串就是一串字符。我们接触的大部分数据都是字符串类型。对字符串进行处理是我们的大部分工作。

### 1. 字符串的定义

在定义字符串时，可以使用单引号、双引号以及三引号。字符串的内容会被包括在两个引号之间。单引号和双引号之间的字符串含义是相同的。如果字符串已经有了单引号，那么我们就需要使用双引号来定义字符串，反之亦然。这里使用的所有引号都应该是 ASCII 中的引号，而不是中文的引号（注意输入法的模式）。例如：

```
>>> "router1's mgmt ip"
"router1's mgmt ip"
```

除了使用不同的引号，我们也可以使用转义字符。例如：

```
>>> 'router1\'s mgmt ip'
"router1's mgmt ip"
```

如果我们需要一个包含换行符的字符串，我们可以使用转义字符“\n”。我们还可以使用三引号。三引号除了出现在字符串的定义中，还会出现在代码的大段描述性文字中，大段的描述性文字也常常作为代码的文档或注释内容。例如：

```
>>>''' R1   10.1.1.1
...     R2   10.1.1.2'''
' R1   10.1.1.1\n    R2   10.1.1.2'
```

### 2. 字符串的合并

刚才讲到了字符串的定义，如何把两个或者多个字符串进行合并呢？我们可以把字符

串接连写出来，这样 Python 会自动对其进行合并。例如：

```
>>> 'R1 ' '10.1.1.1'
'R1 10.1.1.1'
```

但是这种方法用得比较少，通常用于定义变量时。例如：

```
>>> router_info = 'R1 ' '10.1.1.1'
>>> router_info
'R1 10.1.1.1'
```

如果我们现在有两个变量，分别是 hostname 和 ipaddress，合并操作和结果如下：

```
>>> hostname = 'R1'
>>> ipaddress = "10.1.1.1"
>>> hostname + ipaddress
'R110.1.1.1'
```

### 3. 字符串的格式化

在处理字符串的时候，很多时候需要对字符串进行填空操作。这就涉及字符串的格式化问题。在 Python 中，比较常见的方法是使用 “%” 来实现。例如：

```
>>> "R1's ip address is %s" %"10.1.1.1"
'R1's ip address is 10.1.1.1'
```

上面的字符串由两部分组成，第二个 “%” 把字符串分成了前后两个部分，每个部分都是通过一对双引号括起来的。字符串的前一部分 (“R1's ip address is %s”) 是字符串的模板，而后一部分 (“10.1.1.1”) 是模板中需要填的值。其中第一部分的 %s 是转换说明符，它包含如下几个方面的定义。

- ❑ %：表示转换说明符的开始。
- ❑ 转换标志（可选）：其中 “-” 表示左对齐，“+” 表示在值前面加上正负号，空格表示在正数前保留多少个空格，0 表示在数值的位数前可以添加 0。
- ❑ 最小字段宽度（可选）：转换后的字段至少应该有多少个字符宽度。
- ❑ 点 “.” 后加精度（可选）：在转换数值的时候指定小数的精度。
- ❑ 转换类型：见表 8-1（表中只列了部分类型）。

我们来看几个例子：

```
>>> "R1's ip address is %15s" %"10.1.1.1"
'R1's ip address is      10.1.1.1'
```

```
>>> "R1's ip address is %-15s" %"10.1.1.1"
'R1's ip address is 10.1.1.1      '
```

```
>>> "VLAN ID %04d is voice vlan" %10
'VLAN ID 0010 is voice vlan'
```

```
>>> "interface eth1/1 input rate is %0.2f%" %(15.879)
```



```
'interface eth1/1 input rate is 15.88%'
```

```
>>> "interface ge-%d/%d/%d's bandwidth is %dGbps" %(1,2,1,1)
"interface ge-1/2/1's bandwidth is 1Gbps"
```

```
>>>"netmask is 0x%X%X%X%X" %(255,255,254,0)
'netmask is 0xFFFFFE0'
```

表 8-1 字符串的转换类型

转换类型	含 义	转换类型	含 义
d 或 i	带符号的十进制整数	x	不带符号的十六进制数（字母小写）
f	带符号的十进制浮点数	X	不带符号的十六进制数（字母大写）
o	不带符号的八进制数	s	字符串
u	不带符号的十进制数		

#### 4. 字符串的切片处理

字符串也是一个序列，其特性和元组非常相似，我们甚至可以认为字符串就是一个以字符为元素的特殊元组。因此，我们可以使用处理元组一样的方式对字符串进行切片操作。例如：

```
>>> inf="interface GigaEthernet0/0/0/0"
>>> inf[10:]
'GigaEthernet0/0/0/0'
```

字符串 interface 的长度是 9，其实为固定长度，再加上一个空格，则接口名前就有 10 个固定的字符了。因此，为了获取接口名称，我们可以使用 inf[10:] 来获取接口名称。

我们再看一个例子：

```
>>> version="17.1R2.8"
>>> version[:version.index("R")]
'17.1'
```

对于上面的版本信息字符串，“R”字母前为主版本信息，而“R”后为辅版本信息。为了获得主版本信息，我们需要获取“R”前的字符串。方法如下：先使用 index 方法获取到字符“R”的位置信息，然后使用切片的方法来获取主版本信息（为了获取主版本的信息还有其他很多方法，这个例子是其中的一种方法）。

#### 5. 字符串的不可修改性

和元组一样，字符串具备不可修改的特性。虽然我们可以把不同的字符串的值赋给变量。但是这并不意味着字符串发生了改变，而是变量的值发生了改变，我们只是用一个新的字符串替代了原来的字符串而已。读者可以参考 8.2.3 节内容，使用 id 方法检查一下内存中储存字符串的地址的变化情况。

## 6. 字符串的常用方法

### (1) split 方法

字符串的切片处理的第二个例子是获取一个版本信息中的主版本信息，我们还可以通过 split 方法来实现。

```
>>> version
'17.1R2.8'
>>> version.split("R")
['17.1', '2.8']
>>> version.split("R")[0]
'17.1'
```

在这个例子中，我们先使用了 split 方法把字符串分成两个部分，其切分的分割符是字母“R”。如果不指定分割字符或字符串，默认的分割符将为空格。然后，在分割后的列表中取出第一个元素。同样对于第一个例子，我们也可以使用 split 方法。

```
>>> inf
'interface GigaEthernet0/0/0/0'
>>> inf.split()
['interface', 'GigaEthernet0/0/0/0']
>>> inf.split()[1]
'GigaEthernet0/0/0/0'
>>>
```

这里使用了默认的空格作为分割符。

和 split 方法类似的还有 splitlines 方法。关于这个方法，大家可以自行查资料来了解其功能，其参考文档为 <https://docs.python.org/3/library/stdtypes.html>。

### (2) strip 方法

strip 方法用于删除字符串前后一些特定的字符或者字符串。当这个方法没有添加任何参数的时候，默认值为空格。还是上面关于接口的例子，我们希望删除“interface”（注意最后有一个空格，因为空格也是我们不想要的字符）。

```
>>> inf.strip("interface ")
'GigaEthernet0/0/0/0'
```

和 strip 相似的方法还有 lstrip 以及 rstrip。这里笔者留给大家自行去了解它们的功能。除此以外，字符串还有一些其他处理方法。大家可以查相关文档具体了解。



**注意** 由于字符串类型是一个不可变类型，上面的方法都不会修改原变量的值，而是返回了一个值。

## 8.2.5 字典

在 Python 中，字典是一种可变的容器，它可以存储任意的数据对象。有时候字典也被称为哈希表，它是基于键值映射的数据结构。它通过“{key1:value1,key2:value2}”的形式

进行定义。例如：

```
>>> device = {"hostname": "R1", "IP": "1.1.1.1", "Port": 22}
>>> device
{'IP': '1.1.1.1', 'hostname': 'R1', 'Port': 22}
```

其中，key 称为键值，是一个唯一值。为了保证其唯一性，通常采用哈希值方式进行计算。只有那些不可变的值才能进行哈希计算，如数字、字符串以及元组。因此，只有这些值才能作为字典的键值（key），而像列表这样的可变值是不能作为键值的。对于 8.2.3 节留下的疑问，这里可以给出其中一个解释：如果要作为字典的键值，列表是不能够替代元组的。

很显然，由于字典有键值的定义，所以其他每个值的含义更容易被识别和定位。

### 1. 访问字典中的值

字典是通过方括号来获取 key 对应的值的。列表、元组乃至字符串都是通过方括号来获取值的，只不过其方括号中是序列的位置信息，而字典的方括号中是键值。

```
>>> device["hostname"]
'R1'
>>> device["IP"]
'1.1.1.1'
```

除了采用方括号的方式访问字典中的值外，我们还可以通过 get 方法来获取字典中的值。

```
>>> device.get("hostname")
'R1'
>>> device.get("Port")
22
```

这两种方式有什么区别呢？如果使用方括号来获取值，当你所需要访问的键并不存在时，会返回一个 KeyError 的错误。例如：

```
>>> device["host"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'host'
```

而使用 get 方法就不会，甚至还可以返回一个默认值。例如：

```
>>> device.get("port")
>>> device.get("port", 23)
23
```

由于字典的键值是区分大小写的，在上面的代码中并没有获取 port 的值，因为字典 device 中定义的是 Port。

### 2. 修改字典

对字典添加新的内容或者是修改现有的值，都可以通过方括号的方式进行。例如：



```
>>> device
{'IP': '1.1.1.1', 'hostname': 'R1', 'Port': 22}
>>> device["hostname"] = "R2"
>>> device
{'IP': '1.1.1.1', 'hostname': 'R2', 'Port': 22}

>>> device["port"] = 23
>>> device
{'IP': '1.1.1.1', 'hostname': 'R2', 'Port': 22, 'port': 23}
```

如果对一个不存在的键值进行赋值操作，那么将添加一对新的键和值。由于键值是区分大小写的，port 和 Port 不相同，所以字典中多了一个 port 的元素。

### 3. 删除字典中的元素

删除字典中的元素使用 Python 中内置的 del 方法来实现。

```
>>> device
{'IP': '1.1.1.1', 'hostname': 'R2', 'Port': 22, 'port': 23}
>>> del(device['port'])
>>> device
{'IP': '1.1.1.1', 'hostname': 'R2', 'Port': 22}
```

如果键值不存在，将会有抛出异常“KeyError”，有时我们也称之为报错，程序到这里中断。这里是第二次出现异常报错的情况了。我们在学习编程的过程中不要害怕出现错误，每次出现错误时要认真理解这个错误代表什么，这是非常重要的。

```
>>> del(device['port'])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'port'
>>>
```

除了 del 方法以外，pop 方法也可以用来删除一个元素。这个方法和 del 方法的区别在于，删除的内容会作为结果的返回值。同样，当键值不存在的时，会抛出异常“KeyError”。

```
>>> device
{'IP': '1.1.1.1', 'hostname': 'R2', 'Port': 22}
>>> device.pop('Port')
22
>>> device
{'IP': '1.1.1.1', 'hostname': 'R2'}
>>> device.pop('Port')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'Port'
```

### 4. 获取键和值

在上面的这些例子中，我们知道键（key）是什么，然后通过键来获取值。那么我们如何获得一个字典中的所有键呢？我们可以使用 keys 方法来获取所有的键，其他返回值是一

个列表。例如：

```
>>> device
{'IP': '1.1.1.1', 'hostname': 'R2'}
>>> device.keys()
['IP', 'hostname']
```

我们可以使用 `values` 方法来获取所有的值，还可以使用 `items` 方法来获取所有键和值的对应关系。例如：

```
>>> device
{'IP': '1.1.1.1', 'hostname': 'R2'}
>>> device.values()
['1.1.1.1', 'R2']
>>> device.items()
[('IP', '1.1.1.1'), ('hostname', 'R2')]
```

`values` 方法和 `items` 方法返回的都是一个列表，只不过 `items` 方法返回的列表是一个以元组为元素的列表。

## 8.2.6 集合

Python 中的集合和数学中的集合有相似之处，但是 Python 中的集合是一个无序且元素不能重复的集合，其无序、不重复的特性和字典的键是一样的。由于需要做到元素的不重复特性，集合的元素不能是可变数据类型，列表就不可以作为集合的元素。另外，去重是集合的一个功能。我们都知道在数学概念中，集合有交集、并集和补集的概念，在 Python 中，集合也同样有类似的操作。

### 1. 定义

下面先定义两个集合，集合使用 `set` 方法来定义。

```
>>> ip_set1 = set(["1.1.1.1", "1.1.1.2", "1.1.1.5", "10.1.1.1", "10.1.1.10"])
>>> ip_set2 = set(["10.1.1.1", "1.1.1.2", "1.1.1.1", "1.1.1.2", "1.1.1.10"])
```

这里定义的两个集合是一组 IP 地址的信息。我们先来查看一下这两个集合：

```
>>> ip_set1
set(['10.1.1.1', '1.1.1.5', '1.1.1.1', '1.1.1.2', '10.1.1.10'])
>>> ip_set2
set(['10.1.1.1', '1.1.1.10', '1.1.1.1', '1.1.1.2'])
```

`ip_set1` 在一开始定义的时候并没有重复元素，因此还是定义时的那些元素。集合会计算每个元素的哈希值，并通过哈希值来判断是否有重复元素，如果哈希值相同则为相同的元素。很显然，列表和字典无法作为集合的元素。

```
>>> set([[22],[23]])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
TypeError: unhashable type: 'list'
```

```
>>> set([{"port":22}])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

如果一个序列需要成为集合的元素，需要使用元组来代替。

集合 `ip_set2` 在开始定义时有重复元素，在查看变量时 `ip_set2` 已经删除了重复的内容。

## 2. 添加元素

和列表与字典一样，我们可以向集合添加新的元素，方法为 `add` 或 `update`。例如：

```
>>> ip_set1
set(['10.1.1.1', '1.1.1.5', '1.1.1.1', '1.1.1.2', '10.1.1.10'])
>>> ip_set1.add("1.1.1.3")
>>> ip_set1.add("1.1.1.1")
>>> ip_set1
set(['10.1.1.10', '1.1.1.5', '1.1.1.1', '1.1.1.2', '1.1.1.3', '10.1.1.1'])
```

无论被添加的元素是否已经存在，`add` 方法都会执行。

如果需要一次性添加多个元素，需要使用 `update` 方法。

```
>>> ip_set1
set(['10.1.1.10', '1.1.1.5', '1.1.1.1', '1.1.1.2', '1.1.1.3', '10.1.1.1'])
>>> ip_set1.update(["1.1.1.2", "1.1.1.3", "1.1.1.4"])
>>> ip_set1
set(['10.1.1.10', '1.1.1.4', '1.1.1.5', '1.1.1.1', '1.1.1.2', '1.1.1.3', '10.1.1.1'])
```

## 3. 删除元素

删除元素的方法是 `remove`。例如：

```
>>> ip_set1
set(['10.1.1.10', '1.1.1.4', '1.1.1.5', '1.1.1.1', '1.1.1.2', '1.1.1.3', '10.1.1.1'])
>>> ip_set1.remove("1.1.1.2")
>>> ip_set1.remove("1.1.1.2")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: '1.1.1.2'
```

如果需要删除的元素不存在，将会抛出 `KeyError` 异常。关于其他删除元素的方法（如 `discard`、`pop` 以及 `clear` 方法），读者可以查询 <https://docs.python.org> 的相关文档。

## 4. 集合操作

1) 查询当前两个集合状态。例如：

```
>>> ip_set1
set(['10.1.1.10', '1.1.1.4', '1.1.1.5', '1.1.1.1', '1.1.1.3', '10.1.1.1'])
>>> ip_set2
```



```
set(['10.1.1.1', '1.1.1.10', '1.1.1.1', '1.1.1.2'])
>>>
```

2) 取两个集合的交集，即两个集合相同的元素。例如：

```
>>> ip_set1 & ip_set2
set(['10.1.1.1', '1.1.1.1'])
```

3) 取两个集合的并集，即两个集合的和。例如：

```
>>> ip_set1 | ip_set2
set(['1.1.1.4', '1.1.1.5', '1.1.1.1', '1.1.1.2', '1.1.1.3', '10.1.1.1',
'10.1.1.10', '1.1.1.10'])
```

4) 取两个集合的相对补集之和，即并集减去交集的元素，也即两个集合不同元素的内容之和。例如：

```
>>> ip_set1 ^ ip_set2
set(['10.1.1.10', '1.1.1.10', '1.1.1.4', '1.1.1.5', '1.1.1.2', '1.1.1.3'])
```

5) 取集合 A 与 B 的差，即 A 集合删除 AB 交集的元素后留存下来的元素集合。例如：

```
>>> ip_set1 - ip_set2
set(['1.1.1.4', '10.1.1.10', '1.1.1.3', '1.1.1.5'])
```

本节介绍的数据类型是 Python 中最基本的数据类型，使用频率也非常高。本节还介绍了一些常用的方法，当然还有一些其他的方法没有全部覆盖到，这需要大家在日常的编程学习中逐步积累。

## 8.3 基本结构

我们在 7.7 节和 7.8 节中就接触过两种基本结构，即选择结构和循环结构。这两种结构是编程的最小结构单元，在 Python 语言中，这两种结构是不可缺少的。

### 8.3.1 选择结构

在这之前，所有的代码都是自上而下逐条逐行执行的语句。选择结构这部分内容能够将让程序做出选择。在进行选择前，我们需要先了解什么是布尔值。布尔值包含两个值，一个是真 (true)，另一个是假 (false)。在 Python 解释器中，False、None、0、“” (空字符串)、() (空的元组)、[] (空的列表) 以及 {} (空的字典) 等都可以认为是假 (False) 的值。

```
>>> bool(False)
False
>>> bool(None)
False
>>> bool("")
False
>>> bool([])
```

```
False
>>> bool({})
False
>>> bool(())
False
```

这里的布尔值是条件语句需要判断的。条件选择的基本语法如下：

```
if 条件语句:
    语句块
elif 条件语句:
    语句块
else:
    语句块
```



**注意** 上述例子中的空格缩进。空格缩进在 Python 中非常重要，这是初学者最容易忽视的问题。

我们来看一段代码：

```
lab@ubuntu:~$ cat proto.py
#!/usr/bin/python

protocol = input("Please input protocol name: ")
protocol = protocol.lower()

if protocol == "tcp":
    print("TCP's protocol id is 6 ")

elif protocol == "udp":
    print("UDP's protocol id is 17 ")
else:
    print("UNKOWN protocol")
```

说明如下。

第 1 行，“#!/usr/bin/python”表示下面代码的解释器路径。这一点和 Bash 代码是非常类似的。

第 2 行，接受一个输入的值。

第 3 行，把输入的值转化为全小写的字符串。由于在 Python 中，字符串的值是区分大小写的，为了解决字符串的大小写带来的问题，先把所有的字符串都变成了小写。

第 4 行，这里是一个 if 开头的条件判断语句。判断条件是变量 protocol 的值是否等于“tcp”这个字符串。最后需要使用冒号“:”结尾。

第 5 行，print 语句用于向屏幕输出结果。我们需要注意的是，print 语句向内缩进了四个空格，因为 print 语句是 if 这个条件内的语句块。

第 6 行，elif 开头的条件判断语句。这里的判断条件是变量 protocol 是否等于“udp”。

第 7 行，和第 5 行类似。

第 8 行, else 语句, 这里表示当上面的判断都为假时, 执行这个语句。注意, else 后需要使用 “:” 结尾。

第 9 行, 和第 5 行类似。

运行结果如下:

```
lab@ubuntu:~$ python proto.py
Please input protocol name: tcp
TCP's protocol id is 6
lab@ubuntu:~$ python proto.py
Please input protocol name: TCP
TCP's protocol id is 6
lab@ubuntu:~$ python proto.py
Please input protocol name: udp
UDP's protocol id is 17
lab@ubuntu:~$ python proto.py
Please input protocol name: ospf
UNKOWN protocol
```

if 后的条件表达式会用到比较运算符, 表 8-2 给出了常用的比较运算符。

表 8-2 Python 中的比较运算符

运 算 符	说 明	运 算 符	说 明
==	相等	!=	不等于
<	小于	is	两个对象是相同的对象
>	大于	is not	两个对象是不同的对象
<=	小于等于	in	成员关系
>=	大于等于	not in	非成员关系

在表 8-2 中, in 常用于判断某个元素是否属于一个序列。例如:

```
>>> version = "17.1R2.8"
>>> "R" in version
True
```

```
>>> ip_address = ["1.1.1.1", "1.1.1.2", "1.1.1.4"]
>>> "1.1.1.1" in ip_address
True
>>> "1.1.1.3" in ip_address
False
```

运算符 in 可以用于任何序列。上面的例子有字符串和列表, 当然, 元组、字典以及集合都支持运算符 in。

### 8.3.2 循环结构

在 7.8 节中我们就曾提到过, 计算机的特点之一就是能快速重复地做同一件事情。让



计算机重复地完成某项任务就需要用到循环结构。在 Python 语言中，较常用的循环结构为 for 结构和 while 结构。

### 1. for 结构

for 结构的语法如下：

```
for 变量 in 序列：
    语句
```

这里有两个地方需要注意：

- 1) 在 for 这行的最后有“:”；
- 2) 第二行开始的语句有缩进。

7.8 节用 Bash 的 for 结构生成了一段配置。这里我们用 Python 的 for 结构再来实现一次。

```
$ cat bgp_for.py
#!/usr/bin/python
bgp_peers = ["10.1.1.100", "10.1.1.101", "10.1.1.102", "10.1.1.103"]

print("router bgp 100")

for peer in bgp_peers:
    print("neighbor %s remote-as 100" %peer)
    print("neighbor %s update-source lo0" %peer)

print("exit")
```

对比一下 Bash 代码和 Python 代码，我们发现两者基本上是差不多的。

说明（忽略了空行）如下。

第 1 行，指定 Python 解释器的位置。

第 2 行，定义一个列表，列表中为四个 IP 地址。

第 3 行，输出一行配置命令，打印到屏幕上。

第 4 行，for 循环语句。每次循环会从 bgp\_peers 这个列表中取出一个元素并赋值给变量 peer。取元素的顺序是从第 0 个元素开始取，一直到列表的末尾。

第 5~6 行，使用字符串的格式化来输出文本。

第 7 行，打印一个名 exit。

运行结果如下：

```
$ python bgp_for.py
router bgp 100
neighbor 10.1.1.100 remote-as 100
neighbor 10.1.1.100 update-source lo0
neighbor 10.1.1.101 remote-as 100
neighbor 10.1.1.101 update-source lo0
neighbor 10.1.1.102 remote-as 100
neighbor 10.1.1.102 update-source lo0
```

```
neighbor 10.1.1.103 remote-as 100
neighbor 10.1.1.103 update-source lo0
exit
```

## 2. for...else 结构

for...else 结构可以认为是 for 结构的一个增强版。退出 for 结构有三种可能：

- ❑ 正常退出，即遍历了序列中的所有元素后退出；
- ❑ 使用 break 语句在特定条件下强制退出；
- ❑ 在函数中使用 return 语句进行强制退出。

这里的 for...else 语句适合第二种情况，即当正常退出的时候，再执行 else 中的语句，如果通过 break 退出，则不执行 else 内的语句。

假设，我们需要在一个序列中进行查找，并根据查询结果给出相应的输出。我们的代码可以这样写：

```
#!/usr/bin/python
ip_list = ["10.1.1.1", "10.1.1.2", "10.1.1.3", "10.1.1.4"]

IP = input("Please input ip address: ")

for ip in ip_list:
    if ip == IP:
        print("IP address: %s be found in the ip list" % IP)
        break
else:
    print("Can't find the IP address: %s in the ip list" % IP)
```

我们来测试一下代码运行的情况。

```
$ python for_else.py
Please input ip address: 1.1.1.1
Can't find the IP address: 1.1.1.1 in the ip list
```

```
$ python for_else.py
Please input ip address: 10.1.1.1
IP address: 10.1.1.1 be found in the ip list
```

## 3. while 结构

在 Python 中，while 结构的基本语法如下：

```
while 条件判断语句：
    语句
```

如果条件判断语句为真，则执行循环内的语句；如果条件判断语句为假，则结束循环。大部分情况下，while 结构常用于不确定循环次数的情况。下面我们用 while 结构来实现一个简单的菜单选择代码。

```
#!/usr/bin/python
```

```

devices = {"R1": "1.1.1.1",
           "R2": "1.1.1.2",
           "R3": "1.1.1.3",
           "R4": "1.1.1.4"}

while True:
    router_list = devices.keys()
    router_list = sorted(router_list)
    for router_name in router_list:
        print(router_name)

    print("input e to exit ")
    router_input = input("Please select one router: ")
    router_input = router_input.upper()
    if router_input in devices.keys():
        print("I will connect to router %s %s" %(router_input, devices[router_input]))
    elif router_input == "E":
        break
    else:
        print("unknow host")

```

说明如下。

第1~4行，定义了一个字典，字典的键是设备的名称，值是设备的IP地址。

第5行，while语句的开始，其中while的判断语句是True，永远为真，这个循环是永远不会停止的。因此，在while内会有出现break的语句，不然这个程序将永远不会停止。如果你需要的是一个服务端程序，那么它将是一个永远执行下去的程序。

第6~7行，取出字典中的键值，然后排序。

第8~9行，一个for循环，用于输出主机名。虽然这个方法不够简洁，但是它是最容易理解的方式。我们不需要刻意追求代码的简洁性，要更多地关注可读性，尽量用最容易理解的方式来写代码。

第10行，向屏幕输出一行提示。

第11~12行，接受屏幕的输入，并把屏幕的输入内容全部转换为大写。

第13~14行，使用if判断结构，判断输入的字符是否在字典devices的键中。如果在，则输出一段提示内容。这里的提示内容可以替换为其他代码，比如用于连接网络设备进行登录。

第15~16行，判断输入的字符串是否等于“E”或者“e”（第12行把所有的输入都转化为大写，因此这里只需要判断一次大写就可以了），如果是则退出程序。

第17~18行，对于其他任何输入值给出一些信息。

在上面的这个例子中，我们用到了循环嵌套，以及在循环中加入选择结构。其实，再庞大的代码都是由基本结构组成。

测试代码的运行：



```

$ python while.py
R1
R2
R3
R4
input e to exit
Please select one router: r1
I will connect to router R1 1.1.1.1
R1
R2
R3
R4
input e to exit
Please select one router: r3
I will connect to router R3 1.1.1.3
R1
R2
R3
R4
input e to exit
Please select one router: e

```

虽然 Python 没有像 Bash 一样的 select 结构，但是做一个简单的菜单并不复杂。在这里所有的功能都是我们自己来完成的，用到的都是 Python 的基本语句。当我们学习完本章后续的内容后，就可以完成更好的代码。

## 8.4 函数

我们的程序会越写越大，也会越来越复杂。把代码分割成一些小部件，然后把这些小部件组合在一起，这样可以让代码更加清晰和易于维护。

我们可以通过三种方式来分解代码。

第一种是函数 (function)，函数是一组语句的有机组合，可以看成代码的小零件。

第二种是对象 (object)，对象的概念在 8.5 节中会有简单的描述，通常它是对一个功能项的描述，由一些函数和属性来组成。对象可以看成通过一些小零件完成了一个更大的零件模型。

第三种是模块 (module)，模块可以由一个或者多个对象组成，通常它实现的是一个更大集合的功能集。

### 8.4.1 函数的定义

通常而言，函数是用于实现一个功能代码的有序集。我们可以把经常用到的一个功能写成一个函数。函数还可以再嵌入函数，甚至嵌套自己也是可以的。

定义一个函数使用的关键字是 def，后面是函数名。一个函数通常需要有返回值。其语

法如下：

```
def 名称 (参数):           # 如果没有参数可以省略，如果有多个参数需要用逗号分开
    语句
    return 值或表达式      # 不是必需的，但是推荐有返回值
```

我们先来看两个函数，第一个函数用于把 IPv4 的地址转换为一个整数，第二个函数用于把一个整数转换为 IPv4 的点分形式的地址。

```
def ipv4_to_int(ipv4):
    ipv4 = [int(x) for x in ipv4.split(".")]
    ipv4_int = (ipv4[0] << 24) + (ipv4[1] << 16) + (ipv4[2] << 8) + ipv4[3]
    return ipv4_int

def int_to_ipv4(ip_int):
    ipv4 = []
    for x in (24, 16, 8, 0):
        ipv4.append(str(ip_int >> x & 0xFF))
    return ".".join(ipv4)
```

说明如下。

#### (1) 第一个函数

第 1 行，定义了一个名为 `ipv4_to_int` 的函数，函数的命名原则和变量类似（以字母或“\_”开头，后面可以有字母数字和“\_”，并且区分大小写）。这个函数有一个参数是 `ipv4`。

第 2 行，这里出现了一个新的表达式，在 Python 中也叫作列表推导式（也称为列表解析式，list comprehension）。这个表达式和如下的代码是等价的，用这个方法只是让代码更加简洁了。如果不习惯这种列表推导式的方式，也可以用下面的代码进行替换。

```
ip = []
for x in ipv4.split("."):
    x = int(x)
    ip.append(x)
```

第 3 行，位运算（参见 7.5 节相关内容）。Python 中也有相同的运算方法。这里的表达式用到了位左移。

第 4 行，返回函数的值，当函数被调用执行完成后将返回这个值。

#### (2) 第二个函数（下面的行号是第二个函数的行号）

第 1 行，和第一个函数一样，也是一个函数的定义。

第 2 行，定义一个空的列表，用于保存 IP 地址信息。

第 3~4 行，for 循环结构，在表达式中用到了位右移和位与运算（参见 7.5 节相关内容）。其中 `str` 方法用于把数字转换为字符串。

第 5 行，通过字符串的 `join` 方法和列表一起组成一个字符串。这里是 IP 地址的点分方式。

第 6 行，通过 `return` 返回 IP 地址。

这两个函数分别实现了 IP 地址到整数以及整数到 IP 地址的转换。我们可以利用这两个函数来完成对 IP 地址的一些操作。

网络工程师经常会遇到分配 IP 地址的工作。我们在前面的例子中提到如何生成网络配置，其中关于 IP 地址的分配都是预先分配好的。刚才我们已经实现了两个简单的 IP 地址转换的函数。现在，我们可以利用这两个函数来生成 IP 地址相关的配置。代码如下：

```
$ cat func2.py
#!/usr/bin/python
def ipv4_to_int(ipv4):
    ipv4 = [int(x) for x in ipv4.split(".")]
    ipv4_int = (ipv4[0] << 24) + (ipv4[1] << 16) + (ipv4[2] << 8) + ipv4[3]
    return ipv4_int

def int_to_ipv4(ip_int):
    ipv4 = []
    for x in (24, 16, 8, 0):
        ipv4.append(str(ip_int >> x & 0xFF))
    return ".".join(ipv4)

start_ip = "10.1.1.253"
for x in range(0,4):
    ip_address = ipv4_to_int(start_ip) + x * 4
    ip_address = int_to_ipv4(ip_address)
    print("interface gigaEthernet0/1/%d" %x)
    print(" ip address %s 255.255.255.252" %ip_address)
    print(" no shutdown")
```

运行结果如下：

```
$ python func2.py
interface gigaEthernet0/1/0
 ip address 10.1.1.253 255.255.255.252
 no shutdown
interface gigaEthernet0/1/1
 ip address 10.1.2.1 255.255.255.252
 no shutdown
interface gigaEthernet0/1/2
 ip address 10.1.2.5 255.255.255.252
 no shutdown
interface gigaEthernet0/1/3
 ip address 10.1.2.9 255.255.255.252
 no shutdown
```

从运行结果可以看出，IP 地址很好地完成了累加，且很好地实现了 255 的进位操作。

## 8.4.2 函数的参数

前面的例子中的函数使用了一个参数。当然，函数有多个参数也是可以的。实际上，函数并没有限制参数的个数。下面我们通过几个例子来说明 Python 函数参数的几种使用方式。



## 1. 函数的默认值

我们先看下面的代码。

```
def connect(hostname, username="admin", password="admin123", port=23):
    print("connect to %s ...port %d" %(hostname,port))
    print("username is %s" %username)
    print("password is %s" %password)
```

我们在函数 `connect` 参数的定义上对 `username`、`password` 以及 `port` 三个参数提供了预设值。当此函数被调用的时候，如果没有提供这几个参数，系统将会默认使用上面的预设值。在函数体的代码中，我们只是简单地输出了这些变量。

我们现在使用几种方式来给函数传递变量。

**方法一：**我们使用参数名和值的方式来调用函数。

```
connect(hostname="r1",username="netdevops")
```

运行结果如下：

```
$ python func3.py
connect to r1 ...port 23
username is netdevops
password is admin123
```

这里 `hostname` 被赋值为 `r1`，`username` 的值为 `netdevops`，另外两个参数由于在调用时没有提供，因此这里使用了默认值。

**方法二：**我们使用列表来赋值。

```
connect(["r2", "net_admin", "net_admin", 22])
```

运行结果如下：

```
connect to ['r2', 'net_admin', 'net_admin', 22] ...port 23
username is admin
password is admin123
```

在没有改变函数定义的前提下，使用一个列表作为函数的参数进行传递。我们发现，这次运行的结果并没有达到预期。函数把这个列表全部赋值给了第一个参数，后面的参数全部使用了默认值。

现在我们对这个调用做一次修改。修改后的代码如下：

```
connect(*["r2", "net_admin", "net_admin", 22])
```

我们在调用时，在列表前加上了一个“\*”号。这个星号让后面的列表根据顺序的位置依次赋值到了函数中的参数。

**方法三：**我们使用字典来给函数赋值。

```
connect(**{"hostname":"r3", "port": 2202, "username":"net_ops"})
```

这次在调用函数的时候，由于参数来自于一个字典值，我们在字典前面加上了两个

星号。这两个星号代表关键字参数，函数会根据字典中键的名称和函数中参数的名称进行一一的对应。没有提供的参数仍然使用默认值。

这就是我们经常使用的三种传递参数的方法。

## 2. 函数接受冗余参数

在上面的例子中，我们在调用函数的时候，提供的参数都少于函数定义时参数的个数。对于那些没有值传递的参数，函数在调用时将会使用函数定义时的默认值。如果我们传递的参数个数多于函数定义的参数个数，将会出现什么问题呢？我们现在尝试多传递几个参数给函数 `connect`。

```
connect(**{"hostname": "r4",
           "proto": "http",
           "port": 80,
           "username": "netdevops",
           "password": "admin"})
```

运行后我们发现如下问题：

```
Traceback (most recent call last):
  File "func3.py", line 17, in <module>
    connect(**{"hostname": "r4", "proto": "http", "port": 80, "username":
              "netdevops", "password": "admin"})
TypeError: connect() got an unexpected keyword argument 'proto'
```

这里报错是由于出现了原先并没有定义的参数名。

我们现在对之前定义的函数做如下修改：

```
def connect_new(hostname, username, password, port, *args, **kwargs):
    print("connect to %s ...port %d" % (hostname, port))
    print("username is %s" % username)
    print("password is %s" % password)
    print("args: ", args)
    print("kwargs: ", kwargs)
```

在函数中添加了两个参数，它们分别是 “`*args`” 和 “`**kwargs`”。

当调用函数时，“`*args`” 可以接受多余的位置参数，而 “`**kwargs`” 可以接受多余的关键字参数。

现在，再调用一次新的函数 `connect_new`。

```
connect_new("r4", "netdevops", "admin", 2200, "tcp", site="sha")
```

运行结果如下：

```
connect to r4 ...port 2200
username is netdevops
password is admin
('args: ', ('tcp',))
('kwargs: ', {'site': 'sha'})
```

这次运行的结果，除了正常在参数中明确定义四个参数，还有两个是没有被明确定义的变量。没有带名称的参数传递给了 `args`，而带名称的参数则传递给了 `kwargs`。通过这样的参数定义，我们可以很灵活地对函数进行参数传递。

## 8.5 对象

在 8.4 节中，我们了解了函数的一些基本使用情况，函数是把有序的语句整合在一起来实现一些功能的；而对象是把数据和函数都放在一起，通过这样的形式可以改变代码基于过程的逻辑。深刻理解面向对象的编程思想还是需要花费不少时间和功夫的。Python 是全面面向对象的语言，在 Python 中，一切都是对象。

下面我们将简单了解什么是对象，以及怎么创建和使用一个对象。

### 8.5.1 什么是对象

什么是对象？如果我们拿现实世界来类比，对象可以类比为一件东西。东西有其属性，如长、宽、高。我们还可以对东西做很多操作，如推、拉、提等。在编程时，我们也希望能像看待真实世界一样来看待编程的问题。

在理解对象的时候，首先需要理解两个概念，即什么是属性以及什么是方法。对于一台路由器而言，其通常有这些属性：设备的主机名、管理地址、管理的方式（Telnet、SSH 还是 Console）、其运行的路由协议等。对路由器的还有一些常用操作：重新启动、修改配置、保存配置、获取流量信息等。这里提到的主机名、管理地址等都是对象的属性，属性通常是一些值；而操作就是对对象操作的一些方法，这里面有一个动作的执行过程。

除了上面两个在大部分编程书籍中都会重点介绍的概念，还有两个需要大家重点理解的内容：什么是对象的定义和什么是对象的实例。我们再回顾一下上一段对路由器的描述，我们提到路由器会有什么属性和什么方法。这一段描述就是对象的定义，有时我们也称之为对事物的抽象。其所描述的大部分是路由器共有的东西，它是共有东西的一种抽象。现在，假设我们面对一台具体的路由器设备，它的主机名是“R1”，管理地址是“192.168.10.1”，可以通过 Telnet 方式登录，只运行了 OSPF 路由协议。那么，这些内容就是对象的实例了，其针对的是一个具体对象。有时，我们也称为实例化对象。

### 8.5.2 对象的属性和方法

刚才我们用一台路由器举例说明了什么是属性，什么是方法。通常而言，属性的内容是信息，而方法则是动作。

我们在代码中如何表示属性和方法呢？属性的值是数据，数据可以是 Python 内置的基本数据类型，也可以是自己或者第三方定义的对象实例；而方法就是函数。



属性和方法在代码中都是通过“.”来表示的。属性和方法的区别可以通过其最后是否包含“()”来判断。例如：

属性：

```
router.hostname
```

方法：

```
router.get_hostname()
```

我们在写代码时，属性最好使用名词来表示，而方法则用动词或动作短语来表示，这样可以提高代码的可读性。

### 8.5.3 创建对象

定义对象的关键词是 `class`。其语法如下：

```
class 对象名：
```

```
    def 方法名():
        语句
```

我们通常习惯把对象中的函数理解方法。我们可以用上面的语法规则来简单地定义一台路由器：

```
#coding=utf-8
```

```
class Router(object):
```

```
    '''
```

```
    路由器定义。
```

```
    参数为一个字典：
```

```
    {"hostname": "R1",
     "mgmt_ip": "1.1.1.1",
     "username": "admin",
     "password": "admin",
     "mgmt_type": "SSH"}
```

```
    mgmt_type 为设备登录的方式，有 SSH、Telnet、Console 几种方式
```

```
    '''
```

```
    def __init__(self, router_params={}):
```

```
        self.mgmt_ip = router_params.get("mgmt_ip", None)
```

```
        self.hostname = router_params.get("hostname", self.mgmt_ip)
```

```
        self.username = router_params.get("username", None)
```

```
        self.password = router_params.get("password", None)
```

```
        self.mgmt_type = router_params.get("mgmt_type", "SSH")
```

```
    def connect(self):
```

```
        if not self.mgmt_ip:
```

```
            raise AttributeError("you miss mgmt ip address")
```

```
        if not self.username:
```

```
            raise AttributeError("you miss a username")
```

```

if not self.password:
    raise AttributeError("you miss a password")
print("start to connect %s" %self.hostname)

```

说明（下面关于行的序列不包括空行）。

第 1 行，# 开头看似是一个描述，但这里有特殊的意义，即在这个文件中允许出现 utf-8 编码的字符。Python 3 默认支持 unicode 编码格式，其代码中可以出现非 ASCII 编码的字符。但在 Python 2 中就需要添加这一行才可以在代码中使用中文这样的非 ASCII 的字符。我们在下面的描述中包含了中文的注释内容。

第 2 行，这里是类的定义。我们把对象的定义也称为类。在 Python 中，所有对象都是 object 这个类的子类（关于什么是子类，我们会在 8.5.4 节中介绍），这里的 object 是可以省略的。在类的定义最后需要使用“:”结尾。

第 3~12 行，关于这个类的注释，通常使用三引号。这里的注释是可以自动生成类的文档，文档对代码是非常重要的。这里提供了一个简单的例子。这部分一般会包含类属性和方法的说明。

第 13 行，定义一个函数。这个函数名是“\_\_init\_\_”，前后各有两个下划线“\_”。这个函数名是 Python 语言中的一个魔法方法，这个方法会在初始化类的时候被执行。Python 还有很多魔法方法，魔法方法是 Python 语言内置的方法，其代表了一些特殊的意义。关于这些方法，我们可以参考 docs.python.org 中的描述。在类的实例方法中，第一参数是“self”。

第 14~18 行，这里是对类的属性赋值，由于函数的参数是一个字典，因此使用字典的 get 方法来获取相应的值，并且当有不存在的键时给其赋一个默认值。

第 19~26 行，这里是另外一个方法的定义与实现。在这部分中，“raise”用于抛出一个异常，其后面的“AttributeError”是异常的类型。当程序运行的时候，一旦那些必须有值的属性为空，将会中断程序并给出错误的提示。

上面的这个例子是对一个简单类的定义。

## 8.5.4 对象的继承

在现实世界里，人们是通过分类与分层的方式来认知世界的。比如哺乳动物下有犬类，犬类里面有狗，有狼，而狗又可以分为很多品种。在 IT 领域也是一样，一个 IDC 机房里面有服务器和网络设备，网络设备可以分为路由器、防火墙、交换机等。路由器根据厂家和型号又可以分成很多类。我们在定义类的时候也需要用这种层次化的思想来描述其结构。通常把抽象的内容定义为父类，把相对具体的定义为子类，子类和父类之间存在着继承关系。这种继承关系和我们现实生活中的继承关系是非常相似的。

下面我们通过一段代码来解释。

```
class Router(object):
```

```

def __init__(self, params={}):
    self.hostname = params.get("hostname", None)
    self.mgmt_ip = params.get("mgmt_ip", None)

def connect(self):
    print("connect to %s, ip is %s" %(self.hostname, self.mgmt_ip))

class CiscoIOS(Router):

    def __init__(self, params={}):
        super().__init__(params)
        self.vendor = params.get("vendor")

    def goto_enable(self):
        print("goto enable mode")

device_info = {"hostname": "R1",
               "mgmt_ip": "10.1.1.1",
               "vendor": "CiscoIOS"}

device = CiscoIOS(device_info)
print("device.vendor is : ", device.vendor)
device.connect()
device.goto_enable()

```

这段代码大致可以分为三部分。前两个部分定义了两个类，一个是 Router，另一个是 CiscoIOS，其中 CiscoIOS 是 Router 的子类。最后一个部分是对这两个类的测试代码，从变量 device\_info 的定义开始。下面先实例化了 CiscoIOS 类，并且赋值给了变量 device。下面一行代码输出了变量 device 的 vendor 属性，然后调用了 device 的两个方法（函数）。运行结果如下：

```

$ python3 class2.py
device.vendor is : CiscoIOS
connect to R1, ip is 10.1.1.1
goto enable mode

```

我们可以看到，hostname 和 mgmt\_ip 这两个属性代码是在父类（Router）中定义的。方法 connect 也是在父类中定义的。只有 vendor 属性和 goto\_enable 方法是在子类（CiscoIOS）中实现的。这样把一些相同的内容写在父类中，可以减少子类的工作，而且实现了代码的重用。

这一节，我们只是非常简单地介绍了 Python 关于对象的一小部分内容。面向对象编程是一个比较大的话题。对于初学者而言，并不需要马上掌握太多这个方面的内容。但是，有一个初步的理解是非常有必要的。在后续章节的代码中还会用到关于类和对象的知识，笔者会尽可能地给出更加细致的解释。



## 8.6 模块

我们已经介绍了一些 Python 的基础知识。Python 不仅仅核心语言非常强大，还提供了功能丰富的标准库，除此之外，Python 还有非常多的第三方模块（关于第三方模块，我们会在第 10 章中进行更详细的描述）。在本节，我们先对模块有一个初步的认识。

### 8.6.1 什么是模块

任何 Python 程序文件都可以作为模块导入另一个 Python 程序中。模块是比类和函数更大一些的功能组合，可以由一个或者多个文件构成。模块就是一段已经写好的完成一部分特定功能的程序。大多数情况下，在写一个程序时我们并不需要自己来实现每个功能细节，找到合适的模块能帮助我们快速实现想要的功能。

### 8.6.2 如何使用模块

引入一个模块的方法并不复杂。假设我们希望 Python 的程序可以读取命令行中的参数，那么我们可以使用 Python 的标准模块（有时也称为标准库）。

```
#!/usr/bin/python
import sys

args = sys.argv

if len(args)==1:
    print("please input destination IP address")

elif len(args)>1:
    for arg in args[1:]:
        print("Let's test the host, IP is %s" %arg)
```

我们先运行一下上面的程序。

```
$ python module1.py
please input destination IP address
$ python module1.py 1.1.1.1
Let's test the host, IP is 1.1.1.1
$ python module1.py 1.1.1.1 2.2.2.2
Let's test the host, IP is 1.1.1.1
Let's test the host, IP is 2.2.2.2
```

引入一个模块用到的关键字为 `import`，后面为模块的名称。这里我们引入了 `sys` 这个模块。后面的代码中我们就使用了 `sys` 这个对象的一个属性，其保存了命令行中的参数。

第 9 章到第 12 章会介绍更多的第三方模块。我们在进行 NetDevOps 开发时，发现和使用一些好用的第三方模块可以提高我们的开发速度和质量。本书提供的一些第三方模块可以给大家提供一些参考。

## 8.7 小结

本章简单地介绍了 Python 的一些基本语法和经常用到的一些内容。Python 作为一个已经有近 30 年历史的语言，短短一个章节是远远无法涵盖其全部内容的。笔者希望通过这章内容的学习了解能给大家一个总体的感觉。其实 Python 是一门非常容易上手 的语言，通过简单的学习，我们就可以开发一些简单的应用程序，随后就可以一边学习一边开发功能更加丰富的应用。

## 常用数据类型与数据结构定义

一开始，我们提到了 NetDevOps 可以让机器或者程序能够完成更多的工作。为了让程序更加方便地编写和运行，结构化的数据是非常重要的基础元素。在管理和维护网络设备中，哪些是网络设备能提供的常用数据结构，以及这些数据结构是如何定义和描述的，本章会进行较为详细的描述。在描述这些语意和语法的基础上，我们还会介绍部分厂家对它们的支持情况。最后，我们会结合一些实际例子给出一些常用工具和 Python 处理这些数据的方法。

### 9.1 JSON

对于 JSON 这种数据结构，哪怕你并不熟悉，但几乎每天你都在使用。对于 IT 从业人员，这个名词是耳熟能详的，因为它的应用领域确实非常广泛。从图 9-1 可以看出，从 2014 年开始，JSON 的搜索指数呈明显的上升趋势。大量的数据交换使用了 JSON 这种数据格式。下面我们就来了解一下 JSON 这种数据格式的具体情况。

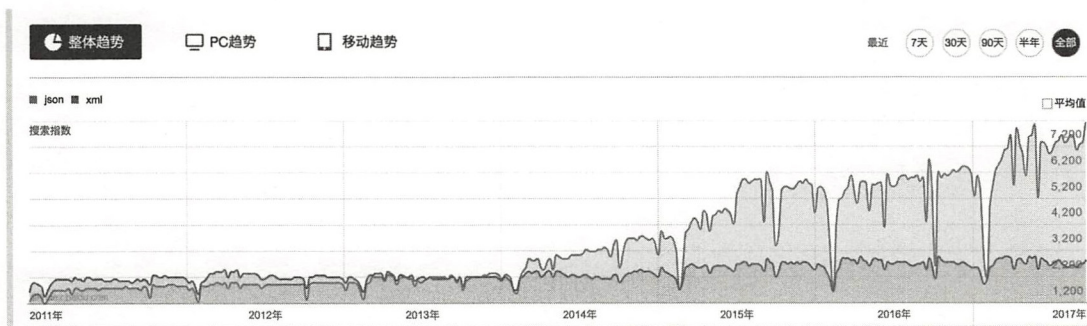


图 9-1 JSON 和 XML 百度指数



### 9.1.1 JSON 简介

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式, 其以文字为基础内容, 兼顾了人和机器的可读性。虽然 JSON 脱胎于 JavaScript, 但已经变成了一个语言无关的数据交互的格式, 这种格式已经被很多的语言所支持。关于 JSON 的内容可以参考 RFC 4627 (<http://www.ietf.org/rfc/rfc4627.txt>)。

JSON 描述的数据结构并不复杂。两个基本概念为键值对 (collection) 和对象 (object)。

键值对由键和值两个部分构成, 键和值之间使用 “:” 进行隔离。其表示方式和第 8 章介绍的 Python 字典的定义非常类似。其实, Python 很容易处理字典和 JSON 格式之间的转换, 我们会在后面介绍如何用 Python 来处理 JSON 数据格式。

其形式如下:

```
{
  key1: value1,
  key2: value2,
}
```

这里键的命名必须是字符串, 而不能是其他内容。值有两种不同的形式: 第一种为简单的值, 其可以是字符串、数值或者布尔值等; 第二种为对象。

对象是由 “{” 所包含的内容, 通常是一个或者几个键值对。几个键值对通过 “,” 隔开。对象和对象之间存在两种序列方式, 一种是无序的, 另一种是有序。无序指的是对象和对象之间没有先后的顺序关系, 这与第 8 章介绍的 Python 集合与字典的元素类似。有序指的是对象之间存在先后的顺序关系, 这与第 8 章介绍的 Python 列表非常类似, 在 JSON 中, 表示顺序关系需要使用 “[ ]” 作为开始符号和结束符号, 里面的元素也通过 “,” 进行分割, 有时也把它称作数组。

对象之间有序的形式:

```
{
  "routers": [
    {"hostname": "R1", "username": "admin"},
    {"hostname": "R2", "username": "netadmin"}
  ]
}
```

从上面的例子来看, 键 “routers” 后面的值是一个有序的数组, 这个数组包含了两个对象。这两个对象分别表示了两台设备的主机名和用户名信息。

图 9-2 是上面例子的树形结构。本章的所有数据结构都是树形结构。树形结构的好处是从树根到树叶有且只有一条路径, 这条路径是不会存在环路的。树形结构不仅仅在数据结构中被常用到, 在很多领域都有应用,

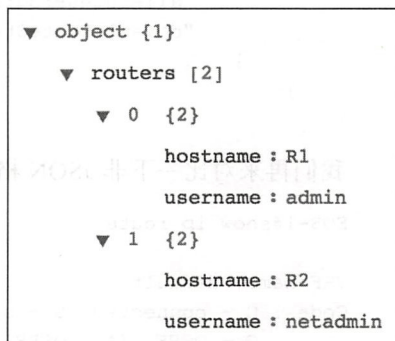


图 9-2 JSON 树形结构

比如网络中的生成树协议（Spanning Tree Protocol, STP）就希望得到一个树状的拓扑，这样就能有效地避免了环路，方便对每个叶节点进行精确定位。另外，树形结构还能清晰地表达出数据的层次含义。

JSON 的数据结构是树形的，不是二维的表结构形式。二维表结构比较典型的有 SQL 数据库、Microsoft Excel 表等。而树形结构在一些 NoSQL 的数据库中也被广泛使用，如 MongoDB。因此，JSON 数据是很容易保存到 MongoDB 中的。因此，读者可以多了解一下 NoSQL 的知识，这里笔者推荐大家先了解一下 MongoDB。

### 9.1.2 网络设备上的 JSON

JSON 数据结构在网络设备上的使用越来越多。例如，Arista EOS 系统就可以直接输出 JSON 数据格式。

```
EOS-1#show ip route | json
{
  "vrfs": {
    "default": {
      "routes": {
        "172.16.1.0/24": {
          "kernelProgrammed": true,
          "directlyConnected": true,
          "routeAction": "forward",
          "vias": [
            {
              "interface": "Ethernet1"
            }
          ],
          "hardwareProgrammed": true,
          "routeType": "connected"
        }
      },
      "allRoutesProgrammedKernel": true,
      "routingDisabled": false,
      "allRoutesProgrammedHardware": true,
      "defaultRouteState": "notSet"
    }
  }
}
```

我们再来对比一下非 JSON 格式的输出：

```
EOS-1#show ip route

VRF name: default
Codes: C - connected, S - static, K - kernel,
        O - OSPF, IA - OSPF inter area, E1 - OSPF external type 1,
        E2 - OSPF external type 2, N1 - OSPF NSSA external type 1,
        N2 - OSPF NSSA external type2, B I - iBGP, B E - eBGP,
```

```

R - RIP, I L1 - ISIS level 1, I L2 - ISIS level 2,
O3 - OSPFv3, A B - BGP Aggregate, A O - OSPF Summary,
NG - Nexthop Group Static Route, V - VXLAN Control Service

```

```
Gateway of last resort is not set
```

```
C      172.16.1.0/24 is directly connected, Ethernet1
```

这种非结构化的输出是我们比较熟悉的形式，但是，大家都清楚基于这种格式进行数据查找和处理还是有一定麻烦。虽然我们在第5章介绍了如何用Linux工具进行这种非结构化文本的处理，但是其对文本输出的依赖性非常强。

我们再看看 Cisco Nexus OS 设备上输出的 JSON 格式。

```
switch# show ip route | json-pretty
```

```

{
  "TABLE_vrf": {
    "ROW_vrf": {
      "vrf-name-out": "default",
      "TABLE_addrf": {
        "ROW_addrf": {
          "addrf": "ipv4",
          "TABLE_prefix": {
            "ROW_prefix": [
              {
                "ipprefix": "10.1.1.0/24",
                "ucast-nhops": "1",
                "mcast-nhops": "0",
                "attached": "true",
                "TABLE_path": {
                  "ROW_path": {
                    "ipnexthop": "10.1.1.1",
                    "ifname": "Eth1/1",
                    "uptime": "PT53S",
                    "pref": "0",
                    "metric": "0",
                    "clientname": "direct",
                    "ubest": "true"
                  }
                }
              }
            ]
          }
        }
      }
    }
  },
  {
    "ipprefix": "10.1.1.1/32",
    "ucast-nhops": "1",
    "mcast-nhops": "0",
    "attached": "true",
    "TABLE_path": {
      "ROW_path": {
        "ipnexthop": "10.1.1.1",
        "ifname": "Eth1/1",
        "uptime": "PT53S",

```



结构，我们可以看出不同的厂家之间存在一定的差异（它们都是已经格式化好的文本格式）。虽然网络设备提供了 JSON 数据格式，但这并不能减少不同厂家甚至是不同的软件版本之间的差异，但提高了数据获取的准确性。

### 9.1.3 JSON-RPC

JSON-RPC (<http://www.jsonrpc.org/specification>) 是一个无状态且轻量级的远程过程调用 (Remote Procedure Call, RPC) 协议。接口的无状态性在第 2 章介绍过, 无状态性降低了交互性, 为分布式处理和异步处理提供了有力的支持。在 9.1.2 节中, 我们看到 Arista EOS 和 Cisco Nexus OS 都支持 JSON 数据格式输出。它们在 API 中也都支持 JSONRPC 的接口规范, 它们分别是 Arista eAPI 和 Cisco NX-API。我们可以使用浏览器进行测试, 图 9-3 和图 9-4 分别为 eAPI 和 NX-API。

### 9.1.4 用 Python 处理 JSON

无论是 Python 2 还是 Python 3，它们都有 JSON 处理模块，我们可以使用 `import` 来导入 JSON 处理模块。

```
>>> import json
```

Python 的 JSON 模块较常用的方法有两个，一个是 `json.dumps`，另一个是 `json.loads`。其中 `json.dumps` 的功能是把 Python 中的对象转化为 JSON 格式的字符串。而 `json.loads` 方法正好相反，用于把 JSON 格式的字符串转化为 Python 中的对象。

下面是两个例子，分别为 `json.dumps` 与 `json.loads`。

```
>>> import json
>>> routers = {"routers": [{"name": "R1", "ip": "10.1.1.1"}, {"name": "R2",
    "ip": "10.1.1.2"}]}
>>> routers_str = json.dumps(routers)
```

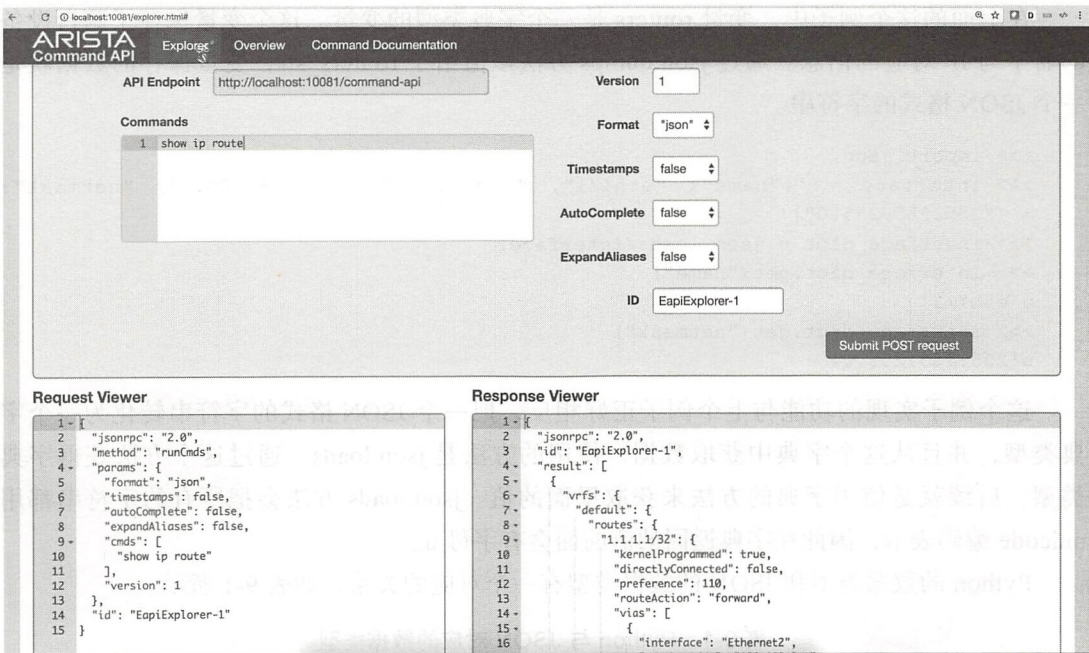


图 9-3 Arista eAPI

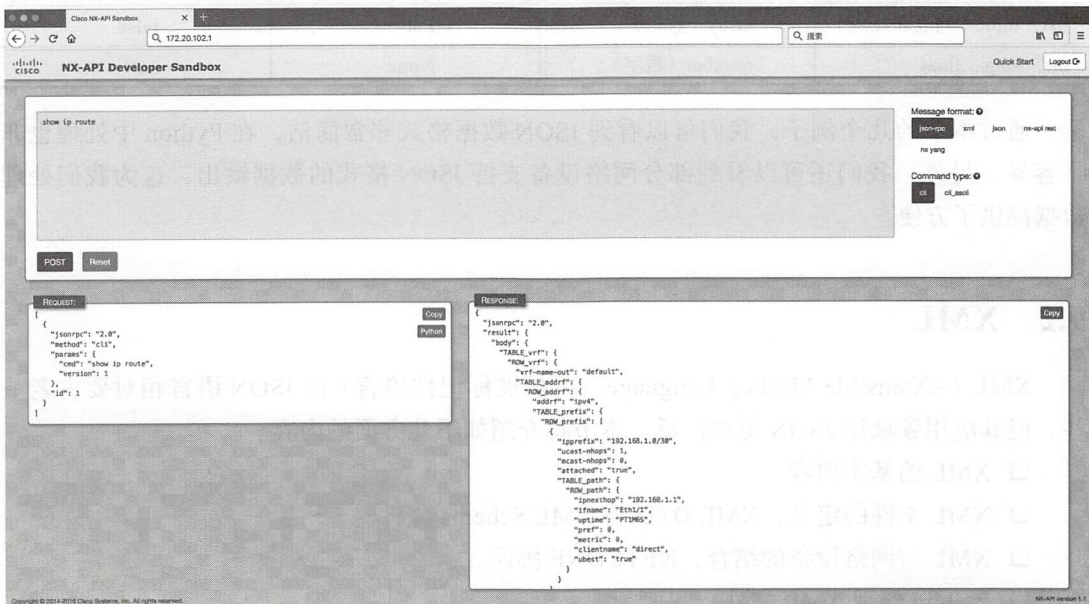


图 9-4 Cisco NXAPI

```
>>> routers_str
```

```
'{"routers": [{"ip": "10.1.1.1", "name": "R1"}, {"ip": "10.1.1.2", "name": "R2"}]}'
```

在上面的这个例子中，变量 `routers` 是一个字典类型的变量。这个变量保存了两台设备的名字与 IP 对应的信息。通过 `json.dumps` 方法赋值给了 `routers_str`，变量保存的数据就是一个 JSON 格式的字符串。

```
>>> import json
>>> interface = '{"name": "eth1/1", "ipaddress": "172.16.100.1", "netmask": "255.255.255.0"}'
>>> interface_dict = json.loads(interface)
>>> interface_dict.get("name")
u'eth1/1'
>>> interface_dict.get("netmask")
u'255.255.255.0'
```

这个例子实现的功能与上个例子正好相反，把一个 JSON 格式的字符串转化为一个字典类型，并且从这个字典中获取数据，使用的方法是 `json.loads`。通过这个方法获得字典类型，后续就是使用字典的方法来获取里面的值。`json.loads` 方法会把所有的字符串都用 `unicode` 编码表示，因此在字典返回的值前面会有字母 `u`。

Python 的数据类型和 JSON 的数据类型有一个对应的关系，如表 9-1 所示。

表 9-1 Python 与 JSON 对应的数据类型

Python	JSON	Python	JSON
dict (字典)	object (对象)	True	ture
list、tuple (列表和元组)	array (数组)	False	false
int、long、float	number (数字)	None	null

通过本节的几个例子，我们可以看到 JSON 数据格式非常简洁，在 Python 中处理也非常容易。另外，我们还可以看到部分网络设备支持 JSON 格式的数据输出，这为我们处理数据提供了方便。

9.2 XML

XML (eXtensible Markup Language，可扩展标记性语言) 比 JSON 语言相对要古老一些，但其应用领域比 JSON 更加广泛。本节将介绍如下几方面的内容。

- ❑ XML 的基本内容。
- ❑ XML 文件的定义，XML DTD 与 XML Schema 文件。
- ❑ XML 与网络设备的结合，NETCONF 协议。
- ❑ Python 处理 XML 文件。

9.2.1 XML 简介

XML 的雏形在 1995 年就形成了，并向 W3C (万维网联盟) 提案，于 1998 年 2 月由



W3C 组织发布为 W3C 的标准 (XML 1.0)。虽然 XML 和 HTML 非常类似,但是它和 HTML 有本质的不同,XML 设计用于数据的传送和存储而并不用于展示数据。有人甚至称 XML 为文本数据库。现在 XML 在很多领域得到了丰富的应用:比如富文本的应用 (Open-Office 的文档采用 XML 文件格式,Microsoft Office 文档也可以采用 XML 文件格式进行保存);再比如网络设备也提供了 XML 的支持,用于管理网络设备。

XML 语法有如下几个特点。

1) 任何起始标签都必须有一个结束标签。例如:

```
<router> R1 </router>
```

2) 对于起始标签和结束标签一起的情况可以采用简化语法。这种语法是在大于号之前使用 “/”。例如:

```
<up/>
```

XML 解析器会将其解析为 <up></up>。

3) 标签必须按照合适的顺序进行嵌套,不能出现交叉嵌套的情况,即标签的包含关系必须是全包含关系,而不能是交集关系。这样定义是为了维持和 JSON 类似的树形结构。例如:

```
<router>
  <name> R1 </name>
  <interfaces>
    <interface>
      <name> ge-0/0/0 </name>
    </interface>
  </interfaces>
</router>
```

错误的格式:

```
<router>
  <name> R1
  <interfaces>
    </name>
    <interface>
      <name> ge-0/0/0 </name>
    </interface>
  </interfaces>
</router>
```

在这个错误的格式中, name 和 interfaces 这两个标记出现了交叉的情况。

4) 在标记内可以添加一个或者多个属性,这是 XML 和 JSON 之间的一个很大区别。属性能让标记提供更多的额外信息。例如:

```
<router vendor="Cisco">
  <name> R1 </name>
```

```

<interfaces>
  <interface>
    <name>GigabitEthernet0/0 </name>
  </interface>
</interfaces>
</router>

```

在这个例子中，router 这个标记中多了一个 vendor 属性。如果我们不使用属性，可以用子标记（有时也称为子元素）。在笔者看来，并没有什么原则规定什么时候使用属性、什么时候使用子标记，我们可以根据自己的需要进行选择，不过在 Python 的数据处理中，也许使用子标记比标记的属性更加容易处理。对于上面的例子，我们使用子标记重新给出 XML 的数据：

```

<router>
  <vendor>Cisco</vendor>
  <name> R1 </name>
  <interfaces>
    <interface>
      <name> GigabitEthernet0/0 </name>
    </interface>
  </interfaces>
</router>

```

### 9.2.2 XML Schema

XML 里面的标记是自己定义的，因此 XML 中的标记是否正确是需要进行校验的。如何来校验这些内容，XML 有两种方式。

第一种是使用 DTD (Documnet Type Definition) 文件。DTD 是一种约束 XML 语言的文档，用于验证 XML 文件，它属于 XML 文件的组成部分。DTD 是一种保证 XML 文档格式正确的有效方法，我们可以通过比较 DTD 文件与 XML 文档来看文档是否符合规范、XML 的标签和属性使用是否正确。

第二种是使用 XSD (XML Schema Definition) 文件，XSD 文件也叫作 XML Schema，其描述了 XML 文档的结构。我们可以用一个 XML Schema 文件来校验一个 XML 文档是否正确，当其文件结构符合要求后才开始读取里面的内容。XML Schema 文件也是通过 XML 的形式进行编写的，这和 DTD 文件不一样，并没有使用新的语法规则。我们可以用 XML 解析器来读取 XSD 文件。现在 XML Schema 已经逐步替代了 DTD 的方式。更多关于 XSD 与 DTD 的信息可以参考 <https://www.ibm.com/developerworks/cn/xml/x-sd/index.html>。

Cisco IOS XR、Juniper JUNOS 都提供 XSD 的文件下载，我们可以通过这些文件来校验与设备交互的 XML 是否符合设备的要求。但是，现在网络设备厂家正在逐步从 XML Schema 转向 YANG 文件（我们会在 9.4 节介绍 YANG 文件的作用），YANG 还会涉及 XSD 文件。由于这些文件几乎提供了所有命令和配置的 XML 文件校验，因此 XSD 文件会非常大。限于篇幅，这里就不提供详细的文件内容了。

9.2.3 NETCONF

NETCONF 协议对于大部分网络工程师而言是既熟悉又陌生的协议。熟悉是因为很多文档中都会提到它，而陌生却是因为在实际的工作中很少用到这个协议。2003 年 5 月，IETF 成立了 NETCONF 工作组，这个工作组提出了基于 XML 的网络配置协议。在 2006 年 12 月通过了 RFC 4741-4744，在 2011 年 6 月又用 RFC 6241、RFC 6242 替代了原来的四个 RFC 文件。在 NETCONF 定义中使用了三种传输协议，分别是 SOAP、BEEP 和 SSH。RFC 6242 更新了基于 SSH 的传输模式，目前使用最为广泛的也是 SSH 协议，其使用 TCP 端口 830 作为其默认的通信端口。由于 XML 可以表达复杂的、模块化的管理对象以及具有内在的逻辑关系等特点，NETCONF 协议完全基于 XML 来表示配置数据和协议消息内容。

NETCONF 现在已经广泛地被网络设备厂家所支持。NETCONF 协议采用了分层的设计结构，和 OSI 网络模型类似，下层为上层提供服务，每一层是对某一个功能的分装。这样的设计既解耦了功能之间的依赖关系，又让其实现起来更加简单。图 9-5 给出了 NETCONF 协议的层次结构，其包括安全通信协议层、消息层、操作层和内容层。

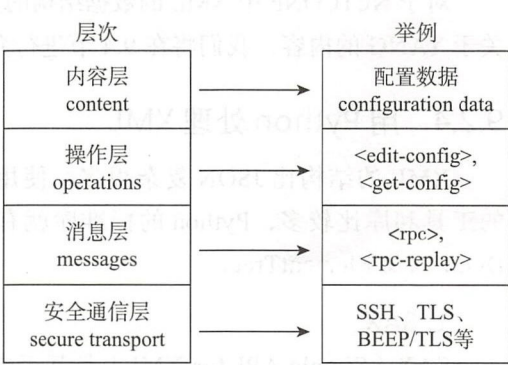


图 9-5 NETCONF 协议的层次结构

1) 安全通信层：这一层提供了服务端和客户端之间的安全通信通道，这部分在 NETCONF 协议中是最底层的定义，也是网络设备最先支持的层次。现在使用 SSH 协议的最为常见，我们在使用 NETCONF 进行通信时，连接网络设备可以使用 SSH 的库来完成底层的通信。

2) 消息层：NETCONF 使用的是 RPC 机制，这里定义了一些简单的 RPC 消息，这些都是通过 XML 标记来体现的，比如 <rpc> 是客户端向服务端发送的 RPC 请求，而 <rpc-reply> 是服务端回复的请求。具体的其他定义可以参考 RFC 5717 的内容。目前支持 NETCONF 协议的网络设备基本都能不错地支持这个层次的内容。这一层定义的内容并不是很多。

3) 操作层：这一层也是基于 XML 标记的，使用 XML 标记定义了很多 RPC 方法。RFC 6241 定义了如下方法，这些方法在 RFC 6241 中都有较为详细的描述。

- ❑ get
- ❑ get-config
- ❑ edit-config
- ❑ copy-config
- ❑ delete-config





- ❑ lock
- ❑ unlock
- ❑ close-session
- ❑ kill-session

4) 内容层：这部分包含了大量和网络设备强相关的数据，里面有设备的配置，也有设备的运行状态的数据等。对于这部分数据结构和标记的定义，每个厂家在具体实现时会有很大的差异。一个网络设备对 NETCONF 的支持是否完善，主要也看这部分的内容是否足够完善。对于网络设备接受或给出的内容，没有充分地完成结构化，那么在使用 NETCONF 进行开发的时候就会遇到非常多的困难和阻力。

对于 NETCONF 中 XML 的数据结构的定义，RFC 6242 使用了新的数据模型语言 YANG。关于 YANG 的内容，我们将在 9.4 节进行介绍。

## 9.2.4 用 Python 处理 XML

XML 的结构比 JSON 复杂很多，使用 Python 处理 XML 时也会复杂很多。处理 XML 的工具和库比较多，Python 的标准库就有三种可以用于 XML 的解析方法，分别是 SAX、DOM 以及 ElementTree。

### 1. SAX

SAX (Simple API for XML) 是基于事件驱动的模式，其通过在解析 XML 过程中触发的事件来处理 XML 文件，在处理事件的时候还可以调用用户定义的回调函数。这种方式对编程语言的能力相对要求高一些，并不太适合编程的初学者使用。这种方式比较适合处理大型文件，这里说的大型文件指的是几百兆字节甚至几十吉字节的文件，这在网络设备环境下的编程并不会经常遇到。

### 2. DOM

DOM (Document Object Model, 文件对象模型) 是 W3C 组织推荐的处理 XML、HTML 等文件的标准编程接口。当 DOM 解释器在解析 XML 文件时，需要一次性把 XML 文件全部读到内存中，然后在内存中将所有的数据都保存在一个树形结构中，然后我们就可以使用 DOM 的一些函数来读取或者修改内存中的树形结构，甚至可以把修改后的内容重新写入 XML 文件。我们现在来看一下用 DOM 处理的例子。这里有一个 XML 文件，其文件名为 interfaces.xml，其内容如下：

```
<router vendor="Cisco">
  <name> R1 </name>
  <interfaces>
    <interface>
      <name>GigabitEthernet0/0</name>
    </interface>
    <interface>
```



```

        <name>GigabitEthernet0/1</name>
    </interface>
</interfaces>
</router>

```

下面我们希望通过代码来获取文件中路由器的厂家的属性以及包含了哪些接口名称。

```

from xml.dom.minidom import parse

domTree = parse("interfaces.xml")
collection = domTree.documentElement
if collection.hasAttribute("vendor"):
    print(collection.getAttribute("vendor"))

interfaces = collection.getElementsByTagName("interface")
for interface in interfaces:
    print(interface.getElementsByTagName("name")[0].firstChild.data)

```

说明如下。

第1行，我们首先引入库。这里使用了 `from...import...` 的方式，这样可以改变引入库的名称空间。如果我们使用 `import xml.dom.minidom.parse`，那么在使用 `parse` 时需要使用 `xml.dom.minidom.parse` 全名。而使用 `from...import...` 就不一样，对于上面的例子，我们后续再使用 `parse` 的时候，只要直接使用就可以了，不用加上 `parse` 前面的内容。

第2行，读取 `interface.xml` 这个文件到内存中。

第3行，获取 `interface.xml` 的根元素。

第4~5行，检查根元素是否包含 `vendor` 属性，如果包含 `vendor` 属性，那么就输出这个属性的值。在这里会输出“Cisco”。

第6~8行，找到元素名为“`interface`”的元素，然后分别输出其子元素为“`name`”的值。运行结果如下：

```

$ python3 dom.py
Cisco
GigabitEthernet0/0
GigabitEthernet0/1

```

关于 `minidom` 标准库的详细内容可以参考 Python 的官方网站：<https://docs.python.org/3.6/library/xml.dom.minidom.html>。

### 3. ElementTree

`ElementTree` 是 Python 独特的 XML 处理方法，它和 DOM 非常类似，不过它可以使用 XPATH 进行搜索。我们还是使用上面的 `interfaces.xml` 文件作为 XML 文档，获取一样的内容。

```

import xml.etree.ElementTree as ET

tree = ET.parse("interfaces.xml")
root = tree.getroot()

```



```
print(root.attrib)

interfaces = root.findall("./*/interface/name")
for interface in interfaces:
    print(interface.text)
```

说明如下。

第 1 行，这里又用了一种导入库的方式：import...as...。在 as 后，我们可以给前面的库取一个别名，这对于比较长的名称比较方便。

第 2 行，导入 interfaces.xml 文件。

第 3 行，获取文件根元素。

第 4 行，输出根元素的属性。

第 5~7 行，通过 xpath 的查找，并输出接口的名称。

运行结果如下：

```
$ python3 et.py
{'vendor': 'Cisco'}
GigabitEthernet0/0
GigabitEthernet0/1
```

在运行结果中，我们看到 ElementTree 获取元素属性的时候，其值是一个字典类型，而不是一个字符串，这和 DOM 库给的结果有一些小区别。更多关于 ElementTree 的文档可以参考 <https://docs.python.org/3/library/xml.etree.elementtree.html>。

综上所述，我们可以知道 XML 比 JSON 有更加丰富的功能，但是也存在一些缺点。表 9-2 为 XML 与 JSON 的比较。进行数据交换时，我们可以根据具体的情况进行选择。在笔者看来，当与设备进行交互时，如果设备支持 JSON 尽量采用 JSON；如果不支持 JSON，使用 XML 也是不错的选择；如果两者都不支持，那么使用半结构化的文本也是可以的；对于自己实现的接口，采用 JSON 作为数据交互的格式也许会更加简洁。

表 9-2 XML 与 JSON 的比较

	XML	JSON
优点	1. 元素具备更多的属性 2. 具有格式校验功能 3. 兼容更多的系统	1. 数据格式简单，兼顾人机读写，占用的带宽小 2. 支持的语言丰富 3. 解析过程消耗资源少
缺点	1. 文件过于冗余庞大 2. 文件格式复杂，不利于人读写 3. 解析文件需要消耗更多的资源	1. 没有 XML 格式通用 2. 没有格式校验机制（只能完成语法检查）

### 9.3 YAML

无论是 JSON 还是 XML 的数据结构，它们都更适合机器用代码来处理，并不是太适合





人进行编写和读取。本节将介绍 YAML，你将了解 YAML 是什么、YAML 适用的场景以及如何用 Python 来处理 YAML 的文本内容。

### 9.3.1 YAML 简介

YAML (Yet Another Markup Language, 另一种标记性语言) 是 Clark Evans 于 2001 年 5 月首次发表的语言，其参考了很多语言的特点，特别是 Python 和 XML 等语言的特点。现在 YAML 的语法规则更新到了 1.2 版本 (第三版)。<http://www.yaml.org> 提供了很多种语言处理 YAML 的开源库，如 C、C++、Ruby、Python、Java、Perl、Golang、PHP 等语言。

#### (1) YAML 的特点

1) YAML 是一种人性化的数据结构定义语言。其语法定义非常简洁，以数据为中心，使用者应更加关心数据。

2) YAML 的语法规则，使得不同的人编写出来的 YAML 文件格式是一致的。在 YAML 的语法中，使用空格与分行作为数据之间的分隔，这就巧妙地避开各种封闭符号的使用。JSON 与 XML 使用封闭符号来完成数据之间的分隔，其中，JSON 使用 “{}” 来完成数据的分隔，而 XML 使用标记对来完成数据的分隔，如 `<rpc></rpc>`。

3) YAML 的文件格式非常适合人的读写习惯。人在读取文本时习惯使用空格、分行与分段来表示文本之间的内在联系，YAML 的语法正好符合人的阅读习惯。

#### (2) YAML 对 XML 的优点

- ☐ YAML 与语言的交互性好。
- ☐ YAML 语法简洁。
- ☐ YAML 解释器容易实现且解析的成本更低。
- ☐ YAML 的可读性更好。

#### (3) YAML 对 JSON 的优点

- ☐ JSON 的语法是 YAML 的子集。
- ☐ YAML 的可读性更好。

#### (4) YAML 的不足

YAML 没有对文本结构的校验机制，即没有 XML 的 Schema 机制。YAML 也没有自己的数据类型的定义，不同的语言可能会解析出不同的数据类型。出于兼容性的考虑，在不同的语言之间进行数据传递时，YAML 并不是一个很好的选择，使用 XML 更加合适。

#### (5) YAML 的主要用途

1) YAML 由于其语法结构简单，解析的成本相对较低，特别适合在脚本语言中使用，比如在 Python、Ruby、Perl、PHP 和 JavaScript 等语言中处理 YAML。

2) YAML 非常适合作为配置文件使用。现在越来越多的工具和平台使用 YAML 作为配置文件，比如 Ansible 的 playbook 文件。OpenStack 中 heat 的模板文件等。

3) YAML 的序列化较为简单且可读性好，可以作为调试过程中的数据输出。



### 9.3.2 YAML 语法

#### 1. 基本语法规则

YAML 是大小写敏感的语言，这和大多数的语言是一致的。

YAML 使用缩进来表示层级关系，在缩进时只能使用空格而不能使用 Tab 键。这一点和 Python 是非常类似的，虽然 Python 可以使用 Tab 键，但是并不推荐使用。对于缩进的空格数目，YAML 并不做严格的要求，只是要求相同层级的元素左侧对齐就可以了。

YAML 使用“#”表示注释，从“#”这个字符开始一直到行尾都为注释部分。

#### 2. 数据类型

YAML 支持的数据格式有以下三种。

- ❑ 纯量：单个的不可再分的值，如一个数值或者字符串等。
- ❑ 数组：一组按照一定次序排列的值，和 JSON 数组是类似的。
- ❑ 对象：为键值对的集合，这和 JSON 的类型也很类似。

##### (1) 纯量

纯量是最基本的不可再分的值。常见的存量有字符串、布尔值、整数、浮点数、时间、日期以及 Null 等。例如：

```
interfaces:
  name: ge-0/0/0
  up: true
```

这里 name 和 up 后面的值就是纯量。

##### (2) 数组

在一组词前使用“-”构成一个数组。例如：

```
- 1.1.1.1
- 1.1.1.2
- 1.1.1.3
```

上面的例子如果用 Python 的数据结构表示就是：

```
["1.1.1.1", "1.1.1.2", "1.1.1.3"]
```

##### (3) 对象

YAML 使用“:”冒号结构来表示对象。例如：

```
hostname: r1
```

##### (4) 复合结构

例如：

```
interfaces:
  - interface: ge-0/0/0
up: true
  - interface: ge-0/0/1
```



```
up: false
- interface: ge-0/0/2
up: true
```

更多关于 YAML 的语法可以参考 <http://www.yaml.org/spec/1.2/spec.html>。

### 9.3.3 用 Python 处理 YAML

Python 语言并没有内置处理 YAML 的模块，如果我们需要处理 YAML 文件，就需要进行额外安装这些模块。<http://www.yaml.org> 网站推荐几个处理 YAML 的 Python 模块，其中最常用的是 PyYAML(<http://pyyaml.org>)，它的源代码位置为 <https://github.com/yaml/py-yaml>。我们将使用这个开源的库来处理 YAML 的内容。我们可以使用 pip 来安装 PyYAML 的模块。

```
$ pip install pyyaml
Collecting pyyaml
  Downloading PyYAML-3.12.tar.gz (253kB)
    100% |=====| 256kB 850kB/s
Installing collected packages: pyyaml
  Running setup.py install for pyyaml ... done
Successfully installed pyyaml-3.12
```

我们先给出一个简单的 YAML 文件，文件名为 interfaces.yaml。

```
interfaces:
- interface: ge-0/0/0
  inet_address: 10.1.1.1/30
- interface: ge-0/0/1
  inet_address: 10.1.1.5/30
- interface: ge-0/0/2
  inet_address: 10.1.1.9/30
```

然后，我们使用 PyYAML 来处理读取这个 YAML 文件的内容。

```
import yaml

with open("interface.yaml") as f:
    f_str = f.read()
    y = yaml.load(f_str)
    for inf in y.get("interfaces"):
        print(inf.get("interface"))
        print(inf.get("inet_address"))
```

说明如下。

第 1 行，我们在代码中引入 PyYAML 的模块，其名称为 yaml。我们只需要导入 yaml 即可。

第 2 行，这里使用了 with 语句。Python 对内建的一些对象进行了改进，加入了对上下文的管理支持。在这里，当 with 语句结束时将自动关闭已经打开的文件。





第 3 行，读取文件的内容。

第 4 行，用 yaml 的 load 方法加载文件的内容。

第 5 行，加载完文件内容后，yaml 模块已经把文本内容转换为一个字典类型的变量。这里通过一个 for 循环来遍历这个字典中的内容。

第 6~7 行，输出字典的内容。

上面的这个例子只是对 YAML 文件做简单的读取处理。如果读者希望了解更多关于 PyYAML 的内容可以参考 <http://pyyaml.org/wiki/PyYAMLDocumentation>。

JSON、XML、YAML 都是数据结构化语言，各具特色，在当前的网络编程中经常遇到。对于每种语言，我们都需要有一定的认识 and 了解。YAML 文件格式非常适合人机共同使用，其较为适用的场景为系统的配置文件和设备配置的纯数据文件。

## 9.4 YANG

YANG 语言和 JSON、XML、YAML 是不同类型的语言，它是一个数据模型语言（data modeling language），常被用在 NETCONF 协议的 RPC（Remote Procedure Calls）和通告（notifications）中。了解和使用 YANG 语言可以有效地提高编程和项目沟通的效率。在本节，你将了解如下几个方面的内容：

- ❑ 什么是 YANG 语言；
- ❑ YANG 语言的基本语法；
- ❑ 如何使用 YANG 语言；
- ❑ 在 Python 中如何使用与处理 YANG 语言。

### 9.4.1 YANG 简介

YANG 在 2008 年 2 月首次发布，目前已经更新到 1.1 版本，其通过 RFC 7950 进行发布。这个语言现在由 IETF NETMOD 工作组进行开发与维护。YANG 是一种数据建模语言，其定义的内容主要用于 NETCONF 协议中内容层的数据模型定义和操作层的数据建模（见图 9-5）。在 NETCONF 协议中，内容层是唯一没有标准化的层，各厂家以及不同的设备之间都存在着一些差异，这就需要一个语言来定义其数据结构以及数据模型。在 YANG 语言出现之前，NETCONF 使用 XML Schema 或 XML DTD 来完成数据的建模，而 YANG 语言更加简洁，可读性更好。其实，YANG 语言的内容可以转化为 XML 的表达方式，这种语言类型被称为 YIN 模型（RFC 6020 中第 11 节有详细的说明）。

什么是数据模型呢？这里通过一个例子来说明。我们先来看一台网络设备的接口配置：

```
root@MX-1# show interfaces ge-0/0/0
mtu 1600;
unit 0 {
    description To-R1-GE0/0/1;
```



```

family inet {
    filter {
        input-list [ filter1 filter2 ];
    }
    address 172.16.0.1/24;
    address 10.1.1.1/24;
}

```

这是 JUNOS 平台上的一个接口配置，其中带下划线的部分是可变内容。对于这个配置，下面我们以 XML 的格式进行输出：

```

<configuration junos:changed-seconds="1509462788" junos:changed-localtime="2017-
10-31 15:13:08 UTC">
    <interfaces>
        <interface>
            <name>ge-0/0/0</name>
            <mtu>1600</mtu>
            <unit>
                <name>0</name>
                <description>To-R1-GE0/0/1</description>
                <family>
                    <inet>
                        <filter>
                            <input-list>filter1</input-list>
                            <input-list>filter2</input-list>
                        </filter>
                    </address>
                    <address>
                        <name>172.16.0.1/24</name>
                    </address>
                    <address>
                        <name>10.1.1.1/24</name>
                    </address>
                </inet>
            </family>
        </unit>
    </interface>
</interfaces>
</configuration>

```

我们可以看到，所有带下划线的值都变成了 XML 元素的值。如果我们把这些值去掉，那么剩下的内容就是这个数据的基本模板，或者也可以看成数据的结构。在这个结构的基础上再加上其数据的类型，那么就可以将其看成一个简单的数据模型了。上面的这个数据结构可以用 YANG 模型来表示。内容如下：

```

module interfaces {
    yang-version "1.1";
    namespace "http://netdevops.cn/yang/configuration/interfaces";
    prefix "nc-if";
}

```





```
revision 2017-10-30 {
    description
        "interface test";
}

typedef ipv4-address-prefix {
    type string {
        pattern '^(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|' +
            '25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4]' +
            '[0-9]|25[0-5])/((([0-9])|([1-2][0-9])|([3][0-2]))$)';
    }
    description
        "ipv4-address";
}

list interface {
    key "name";
    leaf unit {
        type int16 {
            range "0 .. 16384";
        }
    }

    leaf name {
        type string;
    }

    leaf description {
        type string {
            length "1 .. 255";
        }
    }

    leaf mtu {
        type int16 {
            range "64 .. 9200";
        }
    }

    container family {
        container inet {
            container filter {
                leaf-list input-list {
                    type string ;
                }
            }
            list address {
                key "name";
                leaf name {
```





上面的例子自定义了一个 IPv4 地址类型。在 YANG 语言中，需要使用 `typedef` 这个关键字来自定义数据类型。新的数据类型通常是对原有的内置数据类型进行改造而成的，这里的 IPv4 地址类型就是对字符串类型通过正则表达式进行了改造。

```
// 定义 IPv4 的数据类型
typedef ipv4-address-prefix {
    type string {
        pattern '^(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|' +
            '25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4]' +
            '[0-9]|25[0-5])/((([0-9])|([1-2][0-9])|([3][0-2]))$)';
    }
    description
        "ipv4-address";
}
```

2. 节点类型

YANG 定义了一些节点类型，我们在 JSON 和 XML 中也可以看到类似的形式。表 9-4 给出了 YANG 的节点类型。

表 9-4 YANG 的节点类型

名 称	说 明
leaf	叶节点，没有子节点
leaf-list	叶节点，没有子节点，但是里面的值可以存在多个并列的值
list	存在多个类似的并列节点
container	容器节点，只存在子节点，没有具体的值

(1) leaf

leaf 称为叶子节点，和自然界的树叶一样，它是没有子节点的，只有值。比如下面例子中的 `mtu`。我们在定义一个节点的同时，可以定义这个节点的数据类型，这里定义的 `mtu` 是一个整数类型，`mtu` 的取值范围为 64~9200。

```
leaf mtu {
    type int16 {
        range "64 .. 9200";
    }
}
```

这个定义在 XML 结构中表现为

```
<mtu>1600</mtu>
```

我们在 JUNOS 的 XML 文件格式的配置文件中可以看到这种表示方式。

(2) leaf-list

leaf-list 也是一种叶子节点，只不过这种叶节点是一个列表。在下面的例子中，leaf-list



是接口的 filter 内容。在 YANG 中的定义如下：

```
container filter {
    leaf-list input-list {
        type string ;
    }
}
```

这个定义在设备配置的 XML 结构中表现为

```
<filter>
  <input-list>filter1</input-list>
  <input-list>filter2</input-list>
</filter>
```

这样的结构在网络设备的配置中还是很常见的，比如在 BGP 的网络设备配置中，在邻居的策略应用上也会遇到。

### (3) list

list 是列表节点，里面可以存在并列的节点。这样的结构在网络设备的配置中非常多。比如在 9.4.1 节的例子中，接口就是一个典型的 list 节点。一台网络设备会存在很多网络接口，这些接口配置的数据结构是完全一样的，但是其中的值存在一些区别。

在 9.4.1 节的例子中，除了接口节点之外，接口下的 IP 地址也是一个 list 节点。例子中的定义为

```
list address {
    key "name";
    leaf name {
        type ipv4-address-prefix ;
    }
}
```

### (4) container

container 是容器节点，这个节点可以包含任意个、任意类型的子节点。下面例子中的 family 就是一个容器节点，内容如下：

```
container family {
    container inet {
        container filter {
            leaf-list input-list {
                type string ;
            }
        }
        list address {
            key "name";
            leaf name {
                type ipv4-address-prefix ;
            }
        }
    }
}
```



```

    }
  }
}

```

这个数据模型在 XML 结构中表现为

```

<family>
  <inet>
    <filter>
      <input-list>filter1</input-list>
      <input-list>filter2</input-list>
    </filter>
    <address>
      <name>172.16.0.1/24</name>
    </address>
    <address>
      <name>10.1.1.1/24</name>
    </address>
  </inet>
</family>

```

以上就是 YANG 语言的一些基本语法结构。RFC 6020 对 YANG 的语法进行了详细的描述，读者可以参考其更多的语法点，定义出更多、更复杂的数据结构。

### 9.4.3 OpenConfig

如果我们需要使用 NETCONF 的方式对设备进行操作，那么就一定会用到 XML 的数据结构模型。通过这些定义好的数据模型，可以很好地处理设备给出的结构化数据。正如前面提到的，NETCONF 协议对内容层（见图 9-5）并没有给出具体的限制。在使用 NETCONF 来管理设备时，内容层的数据处理确实是一件工作量很大的事。如果有一些定义好的数据模型，那么我们就可以用这些模型来规范我们对设备配置的要求。Open-Config (<http://openconfig.net>) 这个项目是由一些运营商和一些互联网公司组织和参与的开源项目，其目的是能定义出更多的开源数据模型。我们可以在 <https://github.com/openconfig/public> 看到大量的开源数据模型。除了 OpenConfig 项目，IETF 也在 GitHub 上提供了大量的数据模型 (<https://github.com/YangModels/yang>)。

### 9.4.4 Pyang 工具

YANG 文件确实可以定义非常复杂的数据模型。但是，一旦内容多了后，通过人力来检查 YANG 文本的语法错误以及理解它表示的数据模型是比较难的。我们可以使用 Pyang 工具来完成这个工作。Pyang 是一个用 Python 写的 YANG 语法检查和数据转换工具。

我们可以通过 pip 命令来安装 Pyang。

```
$ pip install pyang
```

我们可以使用 Pyang 直接加上 YANG 文件来检查其语法的准确性。例如：

```
$ pyang interfaces.yang
interfaces.yang:6: error: unexpected keyword "revision", expected "prefix"
```

如果文件有问题，Pyang 会给出错误的提示。

我们可以把 YANG 文件的结构定义为以树状形式输出，例如：

```
* $ pyang -f tree interfaces.yang
module: interfaces
  +--rw interface* [name]
    +--rw unit?          int16
    +--rw name            string
    +--rw description?    string
    +--rw mtu?            int16
    +--rw family
      +--rw inet
        +--rw filter
          | +--rw input-list*  string
          +--rw address* [name]
            +--rw name        ipv4-address-prefix
```

我们还可以把数据结构输出到一个 HTML 文件中，然后通过浏览器来查看这个树形结构，如图 9-6 所示。

Module: interfaces, Namespace: http://netdevops.cn/yang/configuration/interfaces, Prefix: nc-if		
Element <a href="#">[+]Expand all</a> <a href="#">[-]Collapse all</a>	Schema	Type
▼ interfaces	module	
▼ interface[name]	list	
unit	leaf	int16
name	leaf	string
description	leaf	string
mtu	leaf	int16
▼ family	container	
inet	container	
filter	container	
input-list	leaf-list	string
address[name]	list	
name	leaf	ipv4-address-prefix

图 9-6 YANG 树形结构

HTML 文件输出的命令如下：

```
$ pyang interfaces.yang -f jstree > interface.html
```

YANG 是一种定义数据结构的建模语言。它不能表示具体的数据，但可以定义具体数据的结构。本节只介绍了 YANG 的基本知识，并没有覆盖 YANG 的全部语法。YANG 语言的功能和作用不局限于本节中的内容。大家可以参考 RFC 或者网站 <http://www.yang-central.org> 来了解更多关于 YANG 的内容。

## 9.5 小结

在开发中，数据的表示和定义是非常重要的环节，本章一共涉及四种与数据相关的语言，前三种是用于直接表示数据的语言，最后一种是用于数据的结构描述和数据建模的语言。不论哪种语言，本章都没能透彻和细致地展开。不过本章给出了很多参考资料的链接，有兴趣的读者可以根据自己的需求进一步了解。



## 第四篇 *Part 4*

# 实 践 篇

通过前面几章的内容，我们已经熟悉了一些基础工具，也学习了 Bash 和 Python 的基本语法。这样我们具备了 NetDevOps 的基本能力。在这一篇中，我们就可以使用这些基础知识来进行一些实践的操作。本篇的内容分为如下几章内容。

第 10 章，使用 Python 和网络设备进行交互。在这一章中，会介绍三种协议和网络设备进行交互。

1) 命令行的方式。这也是网络工程师最常用的方式。

2) NETCONF 的方式。虽然这种方式并不是网络工程师很熟悉的，但是我们可以看到大量的 SDN 控制器在使用它。

3) REST 方式。这种接口形式并不是网络设备最流行的接口类型，不过我们看到支持这种类型的网络设备越来越多，熟悉和了解这种接口是很有意义的。

第 11 章，我们和网络设备进行交互后，就会从网络设备上获取到很多的数据。在这些数据中有很多都是纯文本的、半结构化的数据。如何处理这些文本数据是 NetDevOps 遇到的一个挑战。在这一章中，我们会介绍两个开源的模块来处理这些半结构化的文本。这章的例子中会提供 Cisco IOS 和 Juniper JUNOS 两种风格的文本处理。

第 12 章，处理完设备输出的文本后就可以获得很多的数据。在这些数据中，有两种类型的数据是网络中较为特殊的数据类型，它们分别是网络地址和网络拓扑。这章还会介绍两个开源的模块来处理这些数据类型。

通过这一篇的内容，我们就可以和网络设备进行正常的交互，并对数据进行处理和分析。有了这一篇的基础，我们就可以更好地掌握第五篇的内容。

## 网络设备的连接与登录

我们从第 8 章开始介绍 Python 的内容，在第 9 章我们介绍了三种数据结构语言和一种数据建模语言。在前面两章中，为了实现代码部分的功能，我们用到了一些模块，使用这些现成的模块可以帮助我们快速达成目的。在本章中，我们将介绍连接到网络设备的模块，这些模块都是开源的。为了进行 NetDevOps，我们首先需要实现的是通过代码连接到网络设备（即登录设备）。只有实现了这个功能才能获取设备的信息，从而完成对设备的操作。目前常见的连接设备的方法有三种，分别是传统的模拟登录（其中包括使用 Console、Telnet 和 SSH 方法）、NETCONF 协议登录和 REST API 方式登录。在接下来的章节中，我们将介绍如何使用这三种方法来登录设备，以及会使用到哪些 Python 模块来进行处理。

网络工程师通常会使用 Telnet 或者 SSH 的方式登录网络设备，然后对网络设备进行操作。另外，对于 NETCONF 和 REST 接口方式，我们将分别介绍一个较为常用的模块。

### 10.1 命令行方式登录

网络工程师常通过 Console、Telnet 或 SSH 方式登录网络设备，这是操作网络设备较为传统的方式。几乎所有的网络设备都支持这三种方法，也并不需要网络设备支持特殊的协议。

我们如何使用 Python 来实现这些登录操作呢？可以先想想，如果不使用 Python，我们是如何登录设备的。以 Telnet 登录网络设备为例，我们需要在主机命令行中输入如下命令：

```
$ telnet hostname <port>
```

在输入完上面的命令后，设备系统 OS 会提示我们输入用户名和密码（也存在特殊设

置后不需要用户名和密码就可以直接登录设备的情况), 最后输入网络工程师熟悉的命令来操作网络设备。使用 Python 来完成登录操作其实也是一样的, 通常可以通过两种方式来实现: 一种方式是实现 Telnet 协议; 另一种方式是使用一个伪终端, 在伪终端里面仍然使用传统的命令工具。

下面我们先第一种方式来登录网络设备。在这里介绍如下几个 Python 库:

- ❑ telnetlib
- ❑ paramiko
- ❑ netmiko
- ❑ pexpect

### 10.1.1 telnetlib

telnetlib 是 Python 的一个内置模块, 我们只需要使用 import 命令将其引入, 就可以在程序里面直接调用该模块了。接下来我们用这个模块来登录一台 Cisco IOS 设备。

```

1  # coding:utf-8
2  import telnetlib
3
4  def login(hostname, username, password, port=23):
5
6      tn = telnetlib.telnet(hostname, port=23, timeout=10)
7      tn.set_debuglevel(2)
8      # 输入用户名
9      tn.read_until('Username:')
10     tn.write(username + "\n")
11
12     # 输入登录密码
13     tn.read_until("Password:")
14     tn.write(password + "\n")
15
16     return tn
17
18 if __name__ == "__main__":
19     hostname = "172.20.1.100"
20     username = "cisco"
21     password = "cisco123"
22
23     t = login(hostname, username, password)
24     t.read_until("Router#")
25     t.write("terminal length 0" + "\n")
26     t.read_until("Router#")
27     t.write("show version" + "\n")
28     t.read_until("Router#")
29     t.close()

```

这段代码不是很复杂, 不过比之前的代码增加了一点小技巧。这段代码可以分成三部



分，第一部分是第 1 行和第 2 行，第二部分是第 4~16 行，第三部分是第 18~29 行。

第一部分是这个文件的头部分。

第 1 行，声明这个文件的编码格式是 utf-8 格式，这样就可以在代码中出现中文等非 ASCII 编码的字符。编码格式的声明并不是语句，所以使用了“#”开头。

第 2 行，导入 telnetlib 模块。

第二部分是一个函数的定义。这个函数的功能是登录一台设备。

第 4 行，定义函数 login，关于 Python 函数的定义可以参考 8.4.1 节的内容。

第 6 行，初始化一个 telnet 对象（在 Python 中一切皆对象），telnet 对象初始化最少需要一个参数，这个参数就是主机名。

第 7 行，设置这个对象的 debug 级别，这是跟踪对象的一个方法，这样可以输出和设备交互的详细过程。

第 9 行，等待网络设备显示“Username:”（通过 Telnet 协议发送回来的字符）。

第 10 行，输入用户名，这里需要在用户名这个变量后面加上回车换行符，write 这个方法不会自己添加回车。

第 13~14 行和第 9~10 行是完全类似的，这里就不再重复了。

第 16 行，函数返回 tn 这个对象。这里的 tn 已经是登录成功的一个 Telnet 会话。

第三部分是代码的入口。

第 18 行，采用 Python 的内置变量 \_\_name\_\_，表示当前运行的模块名。如果这个模块被直接运行，那么其值为 \_\_main\_\_；如果作为一个模块被导入并运行后，值就是模块名了。在 Python 中常用这种方式来区分哪些代码是作为模块导入时运行的，而哪些代码是在直接运行的时候执行的。在这个例子中，很显然，第 16 行前的代码是在作为模块时被导入到其他文件中执行的，而第 19 行到结尾的代码是被直接运行的时候才会执行的。

第 19~21 行，定义变量。

第 23 行，使用第 4 行定义的 login 函数，从而获得一个 Telnet 成功登录后的会话（已经登录设备的对象）。

第 24~28 行，代码的含义前面已经讲述过了。只是，这里需要注意的是，在每次输入命令行后，都需要使用 read\_until() 这个方法，否则将会导致上一个命令刚刚传到网络设备上，还没有执行完毕就会去接受下一个命令。特别是在交互式的情况下，这是普遍存在的问题。网络工程师经常会在登录设备后粘贴一些配置到设备上，有时会遇到丢失配置的情况，这就是因为粘贴配置的速度超过了此设备登录会话所能接受的交互能力。所以，我们需要在每输入一条命令时，等待设备执行完这条命令后再去执行后续的命令。

下面是这个程序的执行情况。

```
$ python telnet_ios.py
Telnet(172.20.1.100,23): recv '\xff\xfb\x01\xff\xfb\x03\xff\xfd\x18\xff\xfd\x1f'
Telnet(172.20.1.100,23): IAC WILL 1
Telnet(172.20.1.100,23): IAC WILL 3
```

```

Telnet(172.20.1.100,23): IAC DO 24
Telnet(172.20.1.100,23): IAC DO 31
Telnet(172.20.1.100,23): recv '\r\n\r\nUser Access Verification\r\n\r\nUsername:
\xff\xfc\x01\xff\xfc\x03'
Telnet(172.20.1.100,23): IAC WONT 1
Telnet(172.20.1.100,23): IAC WONT 3
Telnet(172.20.1.100,23): send 'cisco\n'
Telnet(172.20.1.100,23): recv '\xff\xfe\x18\xff\xfe\x1f'
Telnet(172.20.1.100,23): IAC DONT 24
Telnet(172.20.1.100,23): IAC DONT 31
Telnet(172.20.1.100,23): recv 'c'
Telnet(172.20.1.100,23): recv 'is'
Telnet(172.20.1.100,23): recv 'co'
Telnet(172.20.1.100,23): recv '\r\nPassword: '
Telnet(172.20.1.100,23): send 'cisco123\n'
Telnet(172.20.1.100,23): recv '\r\n\r\nRouter#'
Telnet(172.20.1.100,23): send 'terminal length 0\n'
Telnet(172.20.1.100,23): recv 't'
<略>
Telnet(172.20.1.100,23): recv '\r\n'
Telnet(172.20.1.100,23): recv 'Router#'
Telnet(172.20.1.100,23): send 'show version\n'
Telnet(172.20.1.100,23): recv 's'
<略>
Telnet(172.20.1.100,23): recv '\r\n'
Telnet(172.20.1.100,23): recv 'Cisco IOS Software, IOSv Software (VIOS-ADVENTERPR'
<略>
Telnet(172.20.1.100,23): recv 'Read/Write)\r\n0K bytes of ATA CompactFlash 1
(Read/'
Telnet(172.20.1.100,23): recv 'Write)\r\n0K bytes of ATA CompactFlash 2 (Read/
Write'
Telnet(172.20.1.100,23): recv ')\r\n0K bytes of ATA CompactFlash 3 (Read/Write)\r\n\r\n'
Telnet(172.20.1.100,23): recv '\r\n\r\nConfiguration register is 0x0\r\n\r\nRouter#'

```

使用 telnetlib 模块来操作网络设备的逻辑和手动操作设备的方式非常类似。关于 telnetlib 模块的文档可以参考 <https://docs.python.org/3/library/telnetlib.html>。

### 10.1.2 paramiko

telnetlib 模块是使用 Telnet 来登录网络设备的。其实我们在日常工作中更多会使用 SSH 来管理设备，为什么笔者更加推荐使用 SSH 协议，请参考 3.2 节内容。paramiko 是用 Python 语言实现的 SSH 模块，其遵循了 SSH2 的协议内容。Python 是一个跨平台的语言，paramiko 继承了 Python 的这一特点，因此 paramiko 也可以很好地支持跨平台运行，如 Linux、BSD、Windows、Solaris 以及 Mac OS X 等。当我们需要一个跨平台的 SSH 模块时，paramiko 是一个不错的选择。有很多的模块在需要用到 SSH 或 SCP 等功能时，往往也会依

赖这个模块，如 ncclient（NETCONF 模块，详见 10.2.1 节）、Ansible（自动化工具平台）等。这个模块的文档参见 <http://www.paramiko.org>。

对于 Python 而言，第三方模块较常用的安装方法有两种，一种是通过 pip 安装（8.1.3 节已简单的介绍），另一种是通过源代码安装。在有互联网连接的环境下，使用 pip 安装第三方模块是比较简单的方法，pip 会自动下载和安装这个模块所依赖的其他模块。在下面的安装例子中，我们可以看到，pip 下载并安装了 pynacl、cryptography、bcrypt、pyasn1、six、cffi 等模块（在你的安装过程过程中，也许用到的模块会比这个少或多）。更多关于这个模块的安装信息可以参考 <http://www.paramiko.org/installing.html>。

pip 安装命令如下：

```
$ pip install paramiko
Collecting paramiko
  Downloading paramiko-2.3.1-py2.py3-none-any.whl (182kB)
    100% |=====| 184kB 1.5MB/s
Collecting pynacl>=1.0.1 (from paramiko)
  Downloading PyNaCl-1.2.0-cp27-cp27mu-manylinux1_x86_64.whl (696kB)
    100% |=====| 706kB 1.1MB/s
Collecting cryptography>=1.5 (from paramiko)
  Downloading cryptography-2.1.3-cp27-cp27mu-manylinux1_x86_64.whl (2.2MB)
    100% |=====| 2.2MB 445kB/s
Collecting bcrypt>=3.1.3 (from paramiko)
  Downloading bcrypt-3.1.4-cp27-cp27mu-manylinux1_x86_64.whl (57kB)
    100% |=====| 61kB 5.3MB/s
Collecting pyasn1>=0.1.7 (from paramiko)
  Downloading pyasn1-0.3.7-py2.py3-none-any.whl (63kB)
    100% |=====| 71kB 3.5MB/s
Collecting six (from pynacl>=1.0.1->paramiko)
  Downloading six-1.11.0-py2.py3-none-any.whl
Collecting cffi>=1.4.1 (from pynacl>=1.0.1->paramiko)
  Downloading cffi-1.11.2-cp27-cp27mu-manylinux1_x86_64.whl (405kB)
    100% |=====| 409kB 2.0MB/s
Collecting enum34; python_version < "3" (from cryptography>=1.5->paramiko)
  Downloading enum34-1.1.6-py2-none-any.whl
Collecting idna>=2.1 (from cryptography>=1.5->paramiko)
  Downloading idna-2.6-py2.py3-none-any.whl (56kB)
    100% |=====| 61kB 3.5MB/s
Collecting asn1crypto>=0.21.0 (from cryptography>=1.5->paramiko)
  Downloading asn1crypto-0.23.0-py2.py3-none-any.whl (99kB)
    100% |=====| 102kB 4.9MB/s
Collecting ipaddress; python_version < "3" (from cryptography>=1.5->paramiko)
  Downloading ipaddress-1.0.18-py2-none-any.whl
Collecting pycparser (from cffi>=1.4.1->pynacl>=1.0.1->paramiko)
  Downloading pycparser-2.18.tar.gz (245kB)
    100% |=====| 256kB 2.3MB/s
```

安装完 paramiko 后，我们将用这个模块通过 SSH 协议来登录并获取一台 Cisco IOS 设备的配置信息。代码如下：



```

1 # coding:utf-8
2 import getpass
3 import paramiko
4
5 def login(hostname, username, password, port=22):
6     ssh = paramiko.SSHClient()
7     ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
8     ssh.connect(hostname, port, username, password)
9     return ssh
10
11 if __name__ == '__main__':
12     hostname = "172.20.1.100"
13     username = "cisco"
14     password = getpass.getpass()
15     ssh = login(hostname, username, password)
16     stdin, stdout, stderr = ssh.exec_command("show run")
17     for line in stdout.readlines():
18         print(line)

```

在这段代码的第一部分（第 1~3 行），我们除了使用 `paramiko` 这个模块外，还引入了一个 Python 的内置模块 `__getpass`。这个模块用于在命令行中获取密码。在第 14 行会用到这个模块。

第 5~9 行，代码的第二部分。这部分是登录设备的一个函数。

第 6 行，实例化了 `paramiko` 中的 `SSHClient` 这个对象。

第 7 行，设置对 host key 的处理方法。当我们使用 SSH 登录设备时，会获得设备的指纹信息（fingerprint），关于 SSH 指纹的内容可以参考 3.2.3 节的内容。使用 `paramiko` 时也需要处理设备的指纹信息，方法 `set_missing_host_key_policy` 就是用来指定处理指纹信息的策略。默认的策略是 `RejectPolicy`，即拒绝策略。这里使用了自动添加指纹信息的策略。

第 8 行，连接设备的方法。这里需要提供主机名（也可以是 IP 地址）、用户名和密码等信息。

第 11~18 行，代码的第三部分。这部分是代码的入口，也是主要功能的实现部分。

第 11 行，和上一个例子相同，这里不再赘述。

第 12~13 行，定义了两个变量的值。

第 14 行，这里通过 `getpass` 这个模块获取密码信息。通过这个方法来获取密码的时候，你输入的密码是会不会在终端上显示的，这样会提高账号密码的安全性。

第 15 行，调用第二部分定义的 `login` 函数。

第 16 行，在连接到设备后，向设备发送了一个命令。这部分的内容和 3.2.4 节中提到的方式是非常类似的。在这里，`paramiko` 并不会获得一个伪终端，而只能向设备发送一个命令，并得到命令的返回值。之后，就把 SSH 中的这个 session（会话通道）给关闭了。因此，这里只能执行一次命令。

第 17~18 行，输出命令结果。

下面是这个程序的执行情况（省略了部分配置信息）。

```
$ python ssh_ios.py
Password:
Building configuration...
Current configuration : 1335 bytes
!
! Last configuration change at 03:55:02 UTC Tue Nov 7 2017
!
version 15.6
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
!
hostname R1
<略>
aaa new-model
<略>
ip domain name netdevops.cn
username cisco password 0 cisco123
<略>
interface GigabitEthernet0/0
  description "To-R1-G0/0"
  mtu 1600
  ip address 172.20.1.100 255.255.0.0
<略>
line con 0
line aux 0
line vty 0 4
  exec-timeout 40 0
  privilege level 15
  logging synchronous
  transport input all
<略>
end
```

这里介绍的 paramiko 例子是一个比较简单的例子，我们还可以用这个模块来完成 scp 的功能。更多关于 paramiko 的内容可以参考 <http://docs.paramiko.org>。

### 10.1.3 netmiko

前面我们介绍了如何使用 paramiko 来登录网络设备。虽然 paramiko 实现了 SSH2 的功能，但是它并不是专门为网络设备开发的模块。我们在用 paramiko 和网络设备交互时并不是很简单和通用，我们自己还需要处理大量不同厂家设备的差异性所带来的问题。netmiko 是基于 paramiko 开发，专门处理网络设备的 SSH 模块。这个模块目前能支持很多厂家设备的 SSH 连接，其 GitHub 地址是 <https://github.com/kbtyers/netmiko>。

首先，我们可以使用 pip 来安装这个模块。

```

$ pip install netmiko
Collecting netmiko
  Downloading netmiko-1.4.3.tar.gz (47kB)
    100% |=====| 51kB 400kB/s
Requirement already satisfied: paramiko>=1.13.0 in /usr/local/lib/python2.7/dist-packages (from netmiko)
Collecting scp>=0.10.0 (from netmiko)
  Downloading scp-0.10.2-py2.py3-none-any.whl
Collecting pyyaml (from netmiko)
  Downloading PyYAML-3.12.tar.gz (253kB)
    100% |=====| 256kB 1.3MB/s

```

下面我们用 netmiko 模块来实现对一台 Cisco IOS XR 的操作，先列出设备上所有接口的 IP 地址信息，然后在其中一个接口上配置一个 IP 地址，最后获取一次所有接口的信息。

```

1 # coding:utf-8
2 import sys
3 import getopt, getpass
4 from netmiko import ConnectHandler
5
6 def usage():
7     print("Usage: %s [options] " %sys.argv[0])
8     print("-H      : Hostname")
9     print("-u      : Username")
10    print("-p      : Port default is 22")
11    print("-h      : Help informations")
12
13 def handler_opts():
14     cisco_xrv = {
15         'device_type': 'cisco_xr',
16         "port": 22
17     }
18
19     try:
20         opts, args = getopt.getopt(sys.argv[1:], "H:p:u:h", ["hostname",
21             "port", "username", "help"])
22
23         if len(sys.argv) == 1:
24             usage()
25             sys.exit(1)
26         port = 0
27         for opt, arg in opts:
28             if opt in ("-h", "--help"):
29                 usage()
30                 sys.exit(1)
31             elif opt in ("-H", "--hostname"):
32                 cisco_xrv["ip"] = arg
33             elif opt in ("-u", "--username"):
34                 cisco_xrv["username"] = arg
35             elif opt in ("-p", "--port"):
36                 cisco_xrv["port"] = int(arg)

```



```

36     except getopt.GetoptError:
37         usage()
38         sys.exit(1)
39     return cisco_xrv
40
41 if __name__ == '__main__':
42     device = handler_opts()
43     device["password"] = getpass.getpass()
44     net_connect = ConnectHandler(**device)
45     stdout = net_connect.send_command("show ipv4 interface brief")
46     print(stdout)
47
48     config_commands = ["interface GigabitEthernet0/0/0/0", "ipv4 address
        10.1.1.1/24", "no shutdown", "commit"]
49
50     stdout = net_connect.send_config_set(config_commands)
51     print(stdout)
52
53     net_connect.exit_config_mode()
54     stdout = net_connect.send_command("show ipv4 interface brief")
55     print(stdout)

```

这个代码比之前的两个例子都要长一些，但还是只有三部分。

第一部分是第 1~4 行。这部分并没有什么特殊的内容，只是导入了几个模块。其中，`getopt` 模块也是系统内置的模块，专门用于处理命令行中的参数。之前我们的代码都是不带参数的，这次我们需要带上参数。关于 `getopt` 模块的使用，大家可以参考 <https://docs.python.org/2/library/getopt.html>。

第二部分定义了两个函数，这两个函数均用于处理命令行的参数。

第 6~11 行，函数 `usage` 只是输出了这个程序的使用方法。

第 13~40 行，处理命令行的每一个参数的获取。

第三部分是这个程序的主要功能部分（第 42~57 行）。

第 42~44 行，在前面的例子中已经提到过，这里不再赘述。

第 45 行，初始化 `netmiko` 提供的一个类，初始化后，程序就会和网络设备建立一个 SSH 的连接。

第 46 行，向网络设备发送一个命令“`show ipv4 interface brief`”。这里，`send_command` 方法和 `paramiko` 例子中的 `exec_commands` 方法不一样，在 `netmiko` 中，`send_command` 方法在执行完成后并不会关闭此 SSH 会话。

第 47 行，这里会输出第 46 行命令的执行的结果。

第 49 行，定义了一些需要执行的命令的列表。

第 52 行，发送了第 49 行定义的命令。

第 55 行，退出配置模式。退出后可以执行“`show`”命令。

第 56 行，再次执行和第 46 行一样的命令。

下面是这个程序的执行情况。

```
$ python ssh_iosxr.py -H 172.20.1.10 -u admin
Password:
```

```
Tue Nov 7 16:25:48.569 UTC
```

Interface	IP-Address	Status	Protocol	Vrf-Name
MgmtEth0/0/CPU0/0	172.20.1.10	Up	Up	default
GigabitEthernet0/0/0/0	unassigned	Shutdown	Down	default
GigabitEthernet0/0/0/1	unassigned	Shutdown	Down	default
GigabitEthernet0/0/0/2	unassigned	Shutdown	Down	default

```
config term
```

```
Tue Nov 7 16:25:48.859 UTC
```

```
RP/0/0/CPU0:R2(config)#interface GigabitEthernet0/0/0/0
```

```
RP/0/0/CPU0:R2(config-if)#ipv4 address 10.1.1.1/24
```

```
RP/0/0/CPU0:R2(config-if)#no shutdown
```

```
RP/0/0/CPU0:R2(config-if)#commit
```

```
Tue Nov 7 16:25:50.669 UTC
```

```
RP/0/0/CPU0:R2(config-if)#
```

```
Tue Nov 7 16:25:56.319 UTC
```

Interface	IP-Address	Status	Protocol	Vrf-Name
MgmtEth0/0/CPU0/0	172.20.1.10	Up	Up	default
GigabitEthernet0/0/0/0	10.1.1.1	Up	Up	default
GigabitEthernet0/0/0/1	unassigned	Shutdown	Down	default
GigabitEthernet0/0/0/2	unassigned	Shutdown	Down	default

这个模块是基于 paramiko 的，但是对网络设备提供了专门的二次修改和优化，使我们在和网络设备交互时更加方便了。目前这个模块已经得到了很多厂家设备的支持。关于 netmiko 更多的文档，我们可以参考作者的博客 <https://pynet.twb-tech.com/blog>。

### 10.1.4 pexpect

除了使用专门协议方式的模块外，我们还有一种方式（思路）来登录网络设备，即我们可以采用启用 Telnet 或 SSH 子程序并对其进行自动控制的方式来完成与网络设备的交互。这种方式更像人的操作方式。在这里，我们会用到 pexpect 这个模块来完成这个功能。

pexpect 是一个纯 Python 的模块，它可以和很多的 shell 命令进行自动交互，如 telnet、ssh、scp、ftp 等。现在 pexpect 版本已经更新到 4.3，而且这个版本可以同时支持 Python 3 和 Python 2。

和之前介绍的模块一样，可以用 pip 方式来安装 pexpect 模块。

```
$ pip install pexpect
Collecting pexpect
  Downloading pexpect-4.3.0-py2.py3-none-any.whl (55kB)
    100% |=====| 61kB 434kB/s
Collecting ptyprocess>=0.5 (from pexpect)
  Downloading ptyprocess-0.5.2-py2.py3-none-any.whl
Installing collected packages: ptyprocess, pexpect
Successfully installed pexpect-4.3.0 ptyprocess-0.5.2
```

安装完成 pexpect 后，我们用 pexpect 来收集一台 Cisco IOS XR 的接口信息和配置信息。其登录方式为 ssh。代码如下：

```
#!/usr/bin/python
#coding:utf-8
import sys, getpass
import pexpect

# 对从设备获得的结果，按行进行输出
# 在 Python 3 环境下运行，pexpect 的返回值为 byte 类型，可以用 decode 方法将其转成 string 类型
def print_lines(outputs):
    for line in outputs.splitlines():
        print(line.decode("utf-8"))

# 定义一个登录设备的函数
def ssh_login(hostname, username, password, port=22):
    ssh_cmd = "ssh -l %s -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null %s"

    # 使用 pexpect.spawn 方法来创建一个 ssh 子进程
    child = pexpect.spawn(ssh_cmd % (username, hostname))
    # 使用 expect 方法匹配返回结果中的字符串。这里匹配的内容是一个数组，其内容可以是正则表达式。返回
    # 值为所匹配的字符在数组中的编号
    i = child.expect([pexpect.TIMEOUT, "password:"])
    if i == 0:
        print("Connect timeout")
        print_lines(child.before)
        print_lines(child.after)
        sys.exit(1)

    # 如果匹配的是 "password:", expect 返回的结果为 1
    elif i == 1:
        # 使用 sendline 方法，向设备发送密码字符串
        child.sendline(password)
        if child.expect([pexpect.TIMEOUT, "#"]) == 1:
            return child
        else:
            print("timeout")
            sys.exit(1)

    # 定义一个向设备发送命令的函数。这个函数可以被多次使用
    def show_cmd(child, cmd, host_str):
        child.sendline(cmd)
```



```

i = child.expect([pexpect.TIMEOUT, host_str])
if i == 0:
    print("Connect timeout")
    print_lines(child.before)
    print_lines(child.after)
    sys.exit(1)
elif i == 1:
    print_lines(child.before)
    print_lines(child.after)

if __name__ == '__main__':
    hostname = "172.20.1.10"
    username = "admin"
    host_str = "R2#"
    password = getpass.getpass()
    child = ssh_login(hostname, username, password)
    # 由于是伪终端，因此，默认情况下，设备会有分页的停顿
    # 这里需要先输入命令“terminal length 0”
    show_cmd(child, "terminal length 0", host_str)
    # 获取设备的接口信息
    show_cmd(child, "show ip interface brief", host_str)
    # 获取设备的配置信息
    show_cmd(child, "show running", host_str)

```

由于 Windows 平台不支持 Unix 伪终端，因此，这段代码需要在 Linux 或者 MAC OSX 平台下运行。

在上面的例子中，笔者特意把解释写在了代码的注释中，在代码中写注释是一个非常好的习惯，而且注释可以尽量写得详细些。为了在代码中能直接输入中文，我们需要在文件开头定义这个文件的编码格式，这在 telnetlib 模块中也曾经提到过。

这个例子的运行结果如下：

```

$ python ssh_pexpect.py
Password:
terminal length 0

```

```

Fri Nov 10 15:04:00.019 UTC
RP/0/0/CPU0:
R2#
show ip interface brief

```

```

Fri Nov 10 15:04:00.189 UTC

```

Interface	IP-Address	Status	Protocol	Vrf-Name
MgmtEth0/0/CPU0/0	172.20.1.10	Up	Up	default
GigabitEthernet0/0/0/0	10.1.1.1	Up	Up	default
GigabitEthernet0/0/0/1	unassigned	Shutdown	Down	default
GigabitEthernet0/0/0/2	unassigned	Shutdown	Down	default

```

RP/0/0/CPU0:
R2#

```

```

show running

Fri Nov 10 15:04:00.369 UTC
Building configuration...
!! IOS XR Configuration 6.0.1
!! Last configuration change at Fri Nov 10 12:58:14 2017 by admin
!
hostname R2
domain name netdevops.cn
interface MgmtEth0/0/CPU0/0
  ipv4 address 172.20.1.10 255.255.0.0
!
< 略 >
ssh server v2
ssh server vrf default
ssh server netconf vrf default
end

RP/0/0/CPU0:
R2#

```

上面的代码使用 Python 3 也可以很好地运行。代码能同时运行在 Python 2 和 Python 3 环境，并没有想象中那么复杂，毕竟 Python 2 和 Python 3 之间的区别并不是很大。但对于大部分的项目而言，我们只需要选择一个合适的 Python 环境就可以了。

```

$ python3 ssh_pexpect.py
Password:
terminal length 0
Fri Nov 10 15:07:12.776 UTC
RP/0/0/CPU0:
R2#
show ip interface brief
Fri Nov 10 15:07:12.936 UTC

```

Interface	IP-Address	Status	Protocol	Vrf-Name
MgmtEth0/0/CPU0/0	172.20.1.10	Up	Up	default

```

< 略 >
RP/0/0/CPU0:
R2#
show running
Fri Nov 10 15:07:13.126 UTC
Building configuration...
< 略 >
ssh server v2
ssh server vrf default
ssh server netconf vrf default
end
RP/0/0/CPU0:
R2#

```

pexpect 操作设备的方式和人登录设备的操作几乎是一样的。使用 sendlines 方法输入命

令，然后使用 `pexpect` 方法获取设备的返回值。当获取到某个字符串后就认为该设备的输出已经结束，这个字符串通常为设备的主机名加上“#”或“>”。更多关于 `pexpect` 模块的内容，我们可以参考 <http://pexpect.readthedocs.io/en/latest>。`pexpect` 提供的内容并不多，我们可以快速地掌握其方法来和网络设备完成交互式操作。

以上四个模块都是通过传统的命令行方式对网络设备进行管理，这样的操作方式不仅符合网络工程师的习惯，而且在不增加任何其他协议的环境下实现了对网络设备的管理。下面我们将介绍 `NETCONF` 和 `RESTful` 相关的模块。

## 10.2 通过 NETCONF 连接到网络设备

`NETCONF` 协议在 9.2.3 节中已经介绍过，在这里我们使用 `ncclient` 模块来实现 `NETCONF` 和设备的交互。`ncclient` 是一个开源项目，其源码的位置在 <https://github.com/ncclient/ncclient>。本节使用的网络设备为 Juniper vMX 路由器，相应的软件版本为 JUNOS 17.1R2.7。

### 10.2.1 安装 ncclient

我们还是使用 `pip` 来安装 `ncclient` 模块。

```
$ pip install ncclient
Collecting ncclient
  Downloading ncclient-0.5.3.tar.gz (63kB)
    100% |=====| 71kB 481kB/s
Requirement already satisfied: setuptools>0.6 in /usr/lib/python2.7/dist-packages
(from ncclient)
Requirement already satisfied: paramiko>=1.15.0 in /usr/local/lib/python2.7/dist-
packages (from ncclient)
Collecting lxml>=3.3.0 (from ncclient)
  Downloading lxml-4.1.1-cp27-cp27mu-manylinux1_x86_64.whl (5.6MB)
    100% |=====| 5.6MB 171kB/s
< 略 >
Successfully installed lxml-4.1.1 ncclient-0.5.3
```

`ncclient` 依赖的模块有 `paramiko 1.7+`、`lxml 3.3.0+`、`libxml2`、`libxslt`。其中，第一个依赖的模块 `paramiko` 用于完成 SSH 的连接（这个模块在 10.1 节已经介绍过），后面三个模块用于处理 XML。9.2.3 节已介绍过，`NETCONF` 在通信底层最常用的是 SSH，而其上层的数据表示则通常采用 XML 的数据结构。

### 10.2.2 获取配置信息

下面这个例子是使用 `ncclient` 模块来获取设备的配置文件，其返回的结果为 XML 格式的数据。

```
1 #!/usr/bin/env python
```



```

2 # coding:utf-8
3
4 from ncclient import manager
5
6 device = {"host": "172.20.1.11",
7          "port": 830,
8          "username": "lab",
9          "password": "lab123",
10         "hostkey_verify": False,
11         "device_params": {'name': 'junos'}}
12 nc = manager.connect(**device)
13 config = nc.get_config(source="running")
14 print(config)

```

在上面的例子中，第 1 行关于解释器的声明部分和之前例子代码的声明部分有一些小的区别，之前的代码都使用了“/usr/bin/python”，而在这里却使用了“/usr/bin/env python”。前者指定了 Python 解释器在系统中的绝对位置，而后者则希望操作系统能根据环境变量来找到 Python 解释器的位置。在系统存在多版本 Python 共存的情况下，后面这种方式会更好。因此，推荐使用这种写法。

第 11 行，指定了设备的类型。模块 ncclient 能支持多种设备类型，这里列出了一些支持的设备类型：

- ❑ Juniper: device\_params={'name': 'junos'}
- ❑ Cisco CSR: device\_params={'name': 'csr'}
- ❑ Cisco Nexus: device\_params={'name': 'nexus'}
- ❑ Cisco IOS XR: device\_params={'name': 'iosxr'}
- ❑ Cisco IOS XE: device\_params={'name': 'iosxe'}
- ❑ Huawei: device\_params={'name': 'huawei'}
- ❑ Alcatel Lucent: device\_params={'name': 'alu'}
- ❑ H3C: device\_params={'name': 'h3c'}
- ❑ HP Comware: device\_params={'name': 'hpcomware'}
- ❑ Server or anything not in above: device\_params={'name': 'default'}

第 12 行，连接设备。device 是一个字典类型，但是 connect 方法（也称为函数）需要的是单独参数，我们可以通过在字典类型的变量前加两个“\*”来展开字典的内容，用字典的键去匹配方法（函数）中的变量名。

第 13 行，从设备上获取配置信息。其中 get\_config 是 NETCONF 协议定义的标准方法。这段代码的运行结果如下：

```

$ ./netconf.py
<rpc-reply message-id="urn:uuid:6b92a88c-8dd4-492f-bb08-b26656bf9a6d">
  <data>
    <configuration commit-seconds="1510361812" commit-localtime="2017-11-11
00:56:52 UTC" commit-user="root">

```

```

<version>17.1R2.10</version>
<system>
  <host-name>vMX-1</host-name>
[ 略 ]
</system>
<interfaces>
  <interface>
    <name>em0</name>
    <unit>
      <name>0</name>
      <family>
        <inet>
          <address>
            <name>172.20.1.11/16</name>
          </address>
        </inet>
      </family>
    </unit>
  </interface>
</interfaces>
</configuration>
</data>
</rpc-reply>

```

### 10.2.3 获取接口信息

下面这个例子用于获取设备接口信息。首先，我们看看 JUNOS 命令行 CLI 中给出的结果：

```

lab@MX-1> show interfaces terse

```

Interface	Admin	Link	Proto	Local	Remote
ge-0/0/0	up	up			
lc-0/0/0	up	up			
lc-0/0/0.32769	up	up	vp1s		
pfe-0/0/0	up	up			
pfe-0/0/0.16383	up	up	inet inet6		
pfh-0/0/0	up	up			
pfh-0/0/0.16383	up	up	inet		
ge-0/0/1	up	up			

```

< 略 >

```

在 NETCONF 协议中，需要使用 XML 数据格式。JUNOS 命令行其实支持直接获得 XML 的请求内容：

```

lab@MX-1> show interfaces terse | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.1R2/junos">
  <rpc>
    <get-interface-information>
      <terse/>
    </get-interface-information>

```

```

    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

```

元素 `<rpc>` 内的 XML 信息就是向网络设备发送请求信息的关键字（即加粗部分）。为了能很好地处理 XML 的返回值，我们先在命令行中给出部分 XML 的输出内容及格式信息。

```

lab@MX-1> show interfaces terse | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.1R2/junos">
  <interface-information xmlns="http://xml.juniper.net/junos/17.1R2/junos-
    interface" junos:style="terse">
    <physical-interface>
      <name>ge-0/0/0</name>
      <admin-status>up</admin-status>
      <oper-status>up</oper-status>
    </physical-interface>
    <physical-interface>
      <name>lc-0/0/0</name>
      <admin-status>up</admin-status>
      <oper-status>up</oper-status>
      <logical-interface>
        <name>lc-0/0/0.32769</name>
        <admin-status>up</admin-status>
        <oper-status>up</oper-status>
        <filter-information>
          </filter-information>
        <address-family>
          <address-family-name>vpls</address-family-name>
        </address-family>
      </logical-interface>
    </physical-interface>
    [... 省略其他的接口信息 ...]
  </interface-information>
</cli>
  <banner></banner>
</cli>
</rpc-reply>

```

现在，我们修改前一个例子的代码。在 RPC 中发送获取接口信息的 XML 请求，并处理返回后的 XML 信息。

```

1  #!/usr/bin/env python
2  # coding:utf-8
3
4  from ncclient import manager
5
6  device = {"host": "172.20.1.11",
7           "port": 830,
8           "username": "lab",

```



```

9         "password": "lab123",
10        "hostkey_verify": False,
11        "device_params": {'name': 'junos'}}
12 nc = manager.connect(**device)
13
14 interfaces = nc.rpc("<get-interface-information><terse/> </get-interface-infor-
    mation>")
15
16 for interface in interfaces.xpath('//physical-interface/name'):
17     print(interface.text)

```

在代码中，第 14 行使用了 RPC 的方法向 JUNOS 设备发送“get-interface-information”的 XML 请求，然后将 JUNOS 设备返回的数据（XML 格式）赋值给变量 `interfaces`。第 17 行使用了 `xpath`（这个属于 XML 的相关内容）的方式对变量数据进行了筛选。第 18 行输出变量信息里面所有接口的名称。

这部分代码涉及 XML 和 XPATH 的相关内容。关于 XML 的内容，大家可以参考 9.2 节。关于 XPATH 的内容，大家可以参考文档 <http://www.w3school.com.cn/xpath>。

这个代码的运行结果如下：

```

$ python netconf_get_inf.py
ge-0/0/0
lc-0/0/0
pfe-0/0/0
pfh-0/0/0
ge-0/0/1
<略>

```

通过上面这两个例子，我们可以看出使用 `ncclient` 模块来与设备交互的 NETCONF 协议并不太复杂，我们只需要熟悉 NETCONF 协议中的一些常用方法就可以完成对设备的操作。另外，`ncclient` 还集成了对 XML 数据的处理功能，大大方便了对 XML 数据的处理。在具体实践过程中，我们需要先熟悉 NETCONF 协议定义的细节，并结合 `ncclient` 文档就可以完成对网络设备的相关操作。`ncclient` 的文档位置为 <http://ncclient.readthedocs.io/en/latest>。

通过上面的例子我们可以发现，使用 `ncclient` 这个模块和网络设备进行 NETCONF 协议的交互其实并不是很麻烦，甚至比命令行方式（见 10.1 节）更加方便。我们在处理网络设备的信息时，大量的工作是在处理 XML 或 JSON（在上面的例子中没有使用这种数据格式）的数据。因此，掌握更多的 XML 或 JSON 数据格式的处理方法是很有帮助的。

### 10.3 REST

和 NETCONF 协议不一样，REST 协议并不是专门为网络设备定制的协议，它广泛地存在于各种应用系统的 API 中。REST 的英文全称为 Representational State Transfer，通常翻

译为“表现层状态转移”，但从 REST 概念的提出者 Roy Thomas Fielding 在 2000 年的博士论文来看，在 REST 前面需要添加主语“resource”（资源），资源指的就是需要在网络上传送的数据；表现（representational）指的是数据的结构，如 JSON、XML、PNG 等形式的数据；状态转移（state transfer）指的是通过 HTTP 协议来传递信息。因此，REST 的含义是指通过 HTTP 协议传送基于某种数据结构的数据。现在 REST 通常用 JSON 格式来传递文本信息，通信协议也会用到 HTTPS。目前越来越多的网络设备逐步支持 REST 协议，如 Cisco Nexus NX-API、Juniper JUNOS 14.2 之后的软件版本以及大量的 SDN 控制器等。基于 REST 协议，IETF 组织发布了 RESTCONF 协议，其为 RFC 8040（<https://tools.ietf.org/html/rfc8040>）。RESTCONF 是基于 HTTP 的用于网络编程的接口，其数据通过 YANG 模型进行定义，其数据格式可以是 JSON 或 XML 格式。RESTCONF 的出现并不是为了替代 NETCONF 协议。在操作网络设备时，RESTCONF 还是大量地使用了 NETCONF 中定义的一些功能在完成工作。在笔者看来，RESTCONF 也许会让程序员在开发应用时感觉更加舒服。读者并不一定要刻意追求使用 NETCONF 还是 RESTCONF 协议，只要网络设备能提供的，我们都可以去尝试使用，我们只需要找到更加习惯的、能实现期望的功能的协议就可以了。

### 10.3.1 测试 REST 接口

在开始编写代码前，我们可以使用 CURL（见 4.4.3 节）对 REST API（REST 编程接口）进行简单的测试。下面的命令用于获取设备的的路由表信息。

```
$ curl -u "lab:lab123" http://172.20.1.12:8080/rpc/get-route-information
```

```
<route-information xmlns="http://xml.juniper.net/junos/17.1R2/junos-routing"
  xmlns:junos="http://xml.juniper.net/junos/*/junos">
  <!-- keepalive -->
  <route-table>
    <table-name>inet.0</table-name>
    <destination-count>2</destination-count>
    <total-route-count>2</total-route-count>
    <active-route-count>2</active-route-count>
    <holddown-route-count>0</holddown-route-count>
    <hidden-route-count>0</hidden-route-count>
    <rt junos:style="brief">
      <rt-destination>172.20.0.0/16</rt-destination>
      <rt-entry>
        <active-tag>*</active-tag>
        <current-active/>
        <last-active/>
        <protocol-name>Direct</protocol-name>
        <preference>0</preference>
        <age junos:seconds="42766">11:52:46</age>
        <nh>
          <selected-next-hop/>
          <via>fxp0.0</via>
```

```

        </nh>
    </rt-entry>

[ 略 ]

</rt>
</route-table>
</route-information>

```

在 JUNOS 中, REST 资源的格式为 `scheme://device-name:port/rpc/method[@attributes]/params`。其中 `scheme` 可以为 `http` 或 `https`。在 `rpc` 后的 `method` 是指 JUNOS 的 `rpc` 方法, 这些 `rpc` 方法可以通过在 JUNOS CLI 的命令后面加上 “`|display xml rpc`” 来获得, 其获得的方法同样适用于 NETCONF 协议, 前面 NETCONF 的案例中我们已经使用过 `rpc` 方法。



**注意** 这种获取 REST 资源的方法仅限于 JUNOS, 不同厂家的方法并不相同。Cisco Nexus NX-API 的内容可以参考 9.1.3 中的内容。

### 10.3.2 安装 requests 模块

`requests` 模块是 Python 2 和 Python 3 中一个功能较为丰富的基于 HTTP/HTTPS 的客户端模块, 能处理 HTTP/HTTPS 相关的操作。其安装方法和之前的模块类似, 也可以通过 `pip` 进行安装。

```

pip install requests
Collecting requests
  Downloading requests-2.18.4-py2.py3-none-any.whl (88kB)
    100% |=====| 92kB 692kB/s
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Downloading urllib3-1.22-py2.py3-none-any.whl (132kB)
    100% |=====| 133kB 1.5MB/s
Requirement already satisfied: idna<2.7,>=2.5 in /usr/local/lib/python2.7/dist-packages (from requests)
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Downloading chardet-3.0.4-py2.py3-none-any.whl (133kB)
    100% |=====| 143kB 3.4MB/s
Collecting certifi>=2017.4.17 (from requests)
  Downloading certifi-2017.11.5-py2.py3-none-any.whl (330kB)
    100% |=====| 337kB 2.2MB/s
Installing collected packages: urllib3, chardet, certifi, requests
Successfully installed certifi-2017.11.5 chardet-3.0.4 requests-2.18.4 urllib3-1.22

```

### 10.3.3 使用 HTTP get 方法

下面我们以 JUNOS 17.R2 为例, 结合开源的 `requests` 模块来实现对 JUNOS 设备路由表的获取。

```

1 #! /usr/bin/env python
2 # coding:utf-8
3 import getpass

```



```

4 import requests
5 import json
6
7 def get_uri(hostname, method, port=8080):
8     URI = "http://%s:%d/rpc/%s@format=json" %(hostname, port, method)
9     return URI
10
11 if __name__ == '__main__':
12     hostname = "172.20.1.12"
13     username = "lab"
14     password = getpass.getpass()
15     uri = get_uri(hostname, "get-route-information")
16     response = requests.get(uri, auth=(username, password))
17     if response.ok:
18         response_dict = json.loads(response.text)
19         route_information = response_dict.get("route-information")[0]
20         route_table = route_information.get("route-table")[0]
21         rts = route_table.get("rt")
22         for dest in rts:
23             print(dest.get("rt-destination")[0].get("data"))

```

上面代码第7~9行定义了一个生成资源的函数。在 REST API 中，一个资源一般对应唯一的 URI (Uniform Resource Identifiers, 统一资源标示符)。第16行，使用 requests 的 get 方法向设备发送请求。其中参数 auth 是设备登录的用户名和密码。第18行，使用 json.loads 方法把从设备返回的 JSON 数据格式化转换为 Python 的字典类型。而后续的代码用于处理 Python 的字典内容，其目的是只输出路由表中的路由前缀。

运行结果如下：

```

$ python rest.py
Password:
172.20.0.0/16
172.20.1.12/32

```

### 10.3.4 使用 HTTP post 方法

在 HTTP 协议中除了 get 方法还有 post、put 和 delete 等方法。上面例子使用的是 get 方法，我们还可以用 post 方法实现和上面例子一样的功能。其代码如下：

```

#!/usr/bin/env python
# coding:utf-8
import getpass
import requests
import json

if __name__ == '__main__':
    hostname = "172.20.1.12"
    port = 8080
    username = "lab"

```

```

password = getpass.getpass()
s = requests.Session()
s.auth = (username, password)
s.headers.update({"Content-Type": "application/xml"})
s.headers.update({"Accept": "application/json"})
URL = "http://%s:%d/rpc" % (hostname, port)
payload = "<get-interface-information format='json'/>"
response = s.post(URL, data=payload)
r_t = response.text.replace("--nwlrbbmqbhcdarz--", "")
print(r_t)
if response.ok:
    response_dict = json.loads(r_t)
    route_information = response_dict.get("route-information")[0]
    route_table = route_information.get("route-table")[0]
    rts = route_table.get("rt")
    for dest in rts:
        print(dest.get("rt-destination")[0].get("data"))

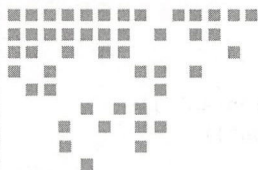
```

这段代码其实并不复杂，读者可以根据前面 get 方法的代码来理解 post 方法的代码。

关于 requests 模块的更多内容，大家可以参考 [http://docs.python-requests.org/zh\\_CN/latest](http://docs.python-requests.org/zh_CN/latest)。

## 10.4 小结

本章介绍了网络设备连接与登录的三种方法，并介绍了每种方法的一些常用模块来实现其基本功能。在这三种方法中，NETCONF 和 REST 通常能获得已经被结构化后的数据。对于 JSON 格式、XML 格式的数据，我们在第 9 章中介绍过数据处理的相关模块。但是，对于最传统的命令行输出格式，我们并没有介绍其处理方法，这部分内容将在第 11 章中进行介绍。



## 命令行文本处理

在成功连接到网络设备后，我们就需要来处理和网络设备之间交互的内容。这些内容可以分为三大类：第一类是传统命令行的输出结果，第二类是网络设备的配置文件，第三类是 JSON 或 XML 格式的数据内容。在第 9 章中，我们介绍了一些 JSON 和 XML 数据的处理方法，在本章中，我们主要侧重于与网络设备的命令行交互以及网络设备的配置文件这两类内容。

前两种类型的数据都是以纯文本的方式体现的，其中，命令行的输出结果（包括设备的运行状态、网络协议的运行情况以及设备的硬件信息等）往往是通过空格、制表符（Tab）以及换行符等字符来对其内容进行格式化存储和显示。网络设备的配置文件又可以分为两大类，一类是以 Cisco 网络设备为代表的，其设备的配置文件除了通过空格、制表符以及换行符等字符进行格式化外，还会结合空格缩进完成配置块的区分。另一类则是 Juniper 的配置文件格式，JUNOS 实际上支持两种形式的配置文件格式，一种是通过空格、换行以及“{}”来格式化和区块化配置文本，还有一种是 set 命令行的形式（具体形式见本章中的例子）。本章将介绍这几种文本的处理方法。

### 11.1 命令行文本输出

在第 5 章中，我们介绍了如何使用 Linux 工具来处理网络设备输出的文本信息。这些工具基本都是行编辑工具，用它们来查找和替换文本内容将会非常方便，但用这些工具来处理网络设备的配置文件时往往不尽如人意，这是因为网络设备的命令行输出结果存在大量的上下文关联。在这里我们介绍一个开源的 Python 模块来处理命令行输出的文本内容，



这个开源库为 TextFSM，这由 Google 开源的一个小项目，其代码托管在 <https://github.com/google/textfsm>。虽然这个模块非常小巧，但是它对于网络设备输出的半结构化文本数据的处理是非常的实用。

### 11.1.1 关于 TextFSM

TextFSM 是一个纯 Python 完成的模块，它是基于模板文件对网络设备输出的半结构化文本数据进行重新格式化的工具，重新格式化后的结果是一个二维的数组（即表格样式）。TextFSM 提供了几个命令行工具用于完成对文本数据的格式化，也可以把它作为一个模块导入 Python 代码中。每处理一个文本内容都将需要一个模板与之来对应。

### 11.1.2 安装 TextFSM

我们依旧可以使用 pip 来安装这个模块。

```
$ pip install textfsm
Collecting textfsm
  Downloading textfsm-0.3.2.tar.gz
Building wheels for collected packages: textfsm
  Running setup.py bdist_wheel for textfsm ... done
  Stored in directory: /root/.cache/pip/wheels/92/7f/36/3dc4b8c2606a92d479b4f9
    86c9deef9c0b293718dd83ace07c
Successfully built textfsm
Installing collected packages: textfsm
Successfully installed textfsm-0.3.2
```

除了使用 pip 的安装方式，TextFSM 也可以通过下载源代码直接安装，我们需要先从 GitHub 下载其源代码，具体命令如下：

```
$ git clone https://github.com/google/textfsm.git
Cloning into 'textfsm'...
remote: Counting objects: 137, done.
remote: Total 137 (delta 0), reused 0 (delta 0), pack-reused 137
Receiving objects: 100% (137/137), 80.19 KiB | 0 bytes/s, done.
Resolving deltas: 100% (71/71), done.
Checking connectivity... done.
```

下载完源代码后，我们进入下载后的目录中，使用 setup.py 安装 TextFSM。

```
$ python setup.py install
running install
running bdist_egg
running egg_info
<略>
Installed /usr/local/lib/python2.7/dist-packages/textfsm-0.3.2-py2.7.egg
Processing dependencies for textfsm==0.3.2
Finished processing dependencies for textfsm==0.3.2
```

11.1.3 TextFSM 模板

首先，我们先看一个例子，通过这个例子希望大家对 TextFSM 模板有一个大概的认识。TextFSM 模板中会用到大量的正则表达式，关于正则表达式的内容可以参考 5.1 节的内容。

下面是一台网络设备 LLDP 邻居结果的输出：

```
R2#show lldp nei
Capability codes:
  (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
  (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID           Local Intf      Hold-time  Capability      Port ID
R4                   Gi0/2          120        R               Gi0/1
R3                   Gi0/1          120        R               Gi0/0
R1                   Gi0/0          120        R               Gi0/1

Total entries displayed: 3
```

对于这个内容，我们通常希望通过表 11-1 的方式来展示相关信息，这个表就是我们通常所说的 A-Z 链路表。这是一张对日常维护工作非常有用的表，其需要获取的内容为上述输出结果的加粗部分。

表 11-1 R2 连接关系表

本段设备名 (A 端)	本段端口 (A 端)	对端端口 (Z 端)	对端设备名 (Z 端)	对端设备类型 (Z 端)
R2	Gi0/2	Gi0/1	R4	R
R2	Gi0/1	Gi0/0	R3	R
R2	Gi0/0	Gi0/1	R1	R

现在我们先给出 TextFSM 模板：

```
Value Filldown LocalHost (\S+)
Value LocalPort (\S+)
Value RemotePort (\S+)
Value RemoteHost (R\S+)
Value DeviceType ([R|B|T|C|W|P|S|O])

Start
  ^${LocalHost}#[>]show lldp
  ^${RemoteHost}\s+${LocalPort}\s+\d+\s+${DeviceType}\s+${RemotePort} -> Record

EOF
```

接着我们来运行一下 TextFSM 自带工具，看看其输出的结果。模板文件名为 lldp\_template，命令行输出的结果命名为 lldp\_neighbor.txt。

```
$ python textfsm.py lldp_template lldp_neighbor.txt
FSM Template:
```

```

Value Filldown LocalHost (\S+)
Value LocalPort (\S+)
Value RemotePort (\S+)
Value RemoteHost (R\S+)
Value DeviceType ([R|B|T|C|W|P|S|O])

Start
  ^${LocalHost} [#|>] show lldp
  ^${RemoteHost} \s+ ${LocalPort} \s+ \d+ \s+ ${DeviceType} \s+ ${RemotePort} -> Record

EOF

FSM Table:
['LocalHost', 'LocalPort', 'RemotePort', 'RemoteHost', 'DeviceType']
['R2', 'Gi0/2', 'Gi0/1', 'R4', 'R']
['R2', 'Gi0/1', 'Gi0/0', 'R3', 'R']
['R2', 'Gi0/0', 'Gi0/1', 'R1', 'R']

```

在输出结果中，textfsm.py 先重新输出了模板的内容，然后给出了格式化后的内容，每一行的输出其实是 Python 的列表格式信息。

#### 11.1.4 如何编写 TextFSM 模板

使用 TextFSM 模板并不需要你有任何的编程能力，但是需要具备定义正则表达式的基础。

##### 1. 简单例子

现在有几个 JUNOS 的版本信息，我们希望获取其中的主版本信息、发布类型、次版本信息以及迭代版本信息。关于 JUNOS 版本信息的定义请参考 [https://www.juniper.net/documentation/en\\_US/junos/topics/reference/general/junos-release-numbers.html](https://www.juniper.net/documentation/en_US/junos/topics/reference/general/junos-release-numbers.html)。

```

17.1R2.7
15.1F6-S6
15.1X53-D231

```

我们可以将上述的三个版本信息分解为如表 11-2 所示的形式。

表 11-2 JUNOS 版本信息

版 本	主 版 本	发 布 类 型	次 版 本	迭 代 版 本
17.1R2.7	17.1	R	2	7
15.1F6-S6	15.1	F	6	S6
15.1X53-D231	15.1	X	53	D231

模板分为三大部分，第一部分为变量的定义，第二部分为匹配规则，第三部分为结束符。每部分需要一个空行进行分隔。



首先，在模板中定义几个变量：

```
Value MainRelease (\d+\.\d)
Value ReleaseType (\w)
Value MinorRelease (\d+)
Value Respin (\w?\d+)
```

其次，添加匹配规则。匹配规则需要由 Start（区分大小写）关键字开始。需要注意规则的格式，特别是空格的格式。在引用变量时，需要把变量写在“\${}”内。

```
Start
    ^${MainRelease}${ReleaseType}${MinorRelease}[\.|\-]${Respin} -> Record
```

最后，添加 EOF。

其完整的模板内容如下：

```
Value MainRelease (\d+\.\d)
Value ReleaseType (\w)
Value MinorRelease (\d+)
Value Respin (\w?\d+)

Start
    ^${MainRelease}${ReleaseType}${MinorRelease}[\.|\-]${Respin} -> Record

EOF
```

下面用 textfsm.py 测试一下结果，获得的 FSM 表如下：

```
FSM Table:
['MainRelease', 'ReleaseType', 'MinorRelease', 'Respin']
['17.1', 'R', '2', '7']
['15.1', 'F', '6', 'S6']
['15.1', 'X', '53', 'D231']
```

## 2. 处理多行的内容

在前面的例子中，所有需要获取的内容都在一行中，如果需要获取的内容分布在多行中，我们也可以方便地获取。

下面是某台设备的接口配置信息。

```
interface GigabitEthernet0/0
    description To_MGT
    ip address 10.1.1.1 255.255.255.0
    duplex auto
    speed auto
!
interface GigabitEthernet0/1
    description To_R2
    ip address 172.16.1.1 255.255.255.252
    duplex auto
    speed auto
    media-type rj45
```

```

!
interface GigabitEthernet0/2
    no ip address
    shutdown
    duplex auto
    speed auto
!
interface GigabitEthernet0/3
    ip address 192.168.100.1 255.255.255.248
    shutdown
    duplex auto
    speed auto
!

```

我们希望通过解析配置文件获得哪些接口配置有 IP 地址。为了实现这个目标，我们的模板文件如下：

```

Value InterfaceName (\S+)
Value IPAddress (\d+\.\d+\.\d+\.\d+)
Value Netmask (\d+\.\d+\.\d+\.\d+)

Start
    ^interface ${InterfaceName}
    ^\s+ip address ${IPAddress} ${Netmask} -> Record

EOF

```



**注意** 符号“->”前后有且只能有一个空格。

运行结果如下：

< 略去模板输出 >

FSM Table:

```

['InterfaceName', 'IPAddress', 'Netmask']
['GigabitEthernet0/0', '10.1.1.1', '255.255.255.0']
['GigabitEthernet0/1', '172.16.1.1', '255.255.255.252']
['GigabitEthernet0/3', '192.168.100.1', '255.255.255.248']

```

在这个例子中，需要记录的信息分布在文本信息的多行中。即使在 interface 和 ip address 之间存在一行接口描述的配置也不会影响所需信息的获取。

### 3. 添加处理流程

在上述的配置文件中，我们获取了所有接口的 IP 地址，即使是 shutdown 接口的配置也获取了。如果我们希望只获取那些被 shutdown 接口的 IP 地址的信息呢？我们可以在模板中加入一些流程控制来实现这样的目的。

```

Value InterfaceName (\S+)
Value IPAddress (\d+\.\d+\.\d+\.\d+)
Value Netmask (\d+\.\d+\.\d+\.\d+)

```

```

Value Shutdown (shutdown)

Start
  ^interface ${InterfaceName}
  ^\s+ip address ${IPAddress} ${Netmask} -> Shut

Shut
  ^\s+${Shutdown} -> Record
  ^\! -> Start

EOF

```

运行结果如下：

```

FSM Table:
['InterfaceName', 'IPAddress', 'Netmask', 'Shutdown']
['GigabitEthernet0/3', '192.168.100.1', '255.255.255.248', 'shutdown']

```

在这个模板中，我们先获取了接口名和 IP 地址信息。如果两者都能成功获取，则会跳转到另一个信息查找块。在信息查找块中，先查找是否存在 shutdown 的信息，如果可以正常查找到，就记录下这条信息。然后查找一个接口配置块的结尾信息（在 Cisco 的配置中，结尾信息是“!”字符，这个字符既可以用于注释，也会表示一段配置的结束），如果找到结尾信息符就会结束本接口的查找，并开始进行一个新的接口的查找。因此，最后我们得到的结果就是所有 shutdown 的接口的 IP 地址信息。

#### 4. 使用其他动作

每行匹配规则后都有一个动作（在“->”后定义的内容）。其格式如下：

行动作，记录动作 状态转换

行动作包括如下两个动作。

- ❑ Next：这是每一行的默认动作，用于读取一行输入的文本（需要匹配的文本），并从 Start 或者自己定义的匹配块（如上一个例子中的 Shut 块）开始重新进行规则匹配。
- ❑ Continue：保持当前的输入文本行，进行下一行的规则匹配。

记录动作包括如下四个动作。

- ❑ NoRecord：默认行为，不记录当前匹配的值。
- ❑ Record：记录之前匹配的值。
- ❑ Clear：清除匹配的值，但不清除这些在变量中定义为 Filldown 的值。
- ❑ Clearall：清除所有匹配的值。

状态转换：在这里我们可以把规则指到另一个匹配块，和前面的动作之间使用一个空格进行分隔。

基于上一个例子，我们如何只查找没有被 shutdown 的接口信息呢？在 IOS 中，“no shutdown”的配置并不体现在配置中。我们可以对配置模板做如下修改：



```

Value InterfaceName (\S+)
Value IPAddress (\d+\.\d+\.\d+\.\d+)
Value Netmask (\d+\.\d+\.\d+\.\d+)
Value Shutdown (shutdown)

Start
    ^interface ${InterfaceName}
    ^\s+ip address ${IPAddress} ${Netmask} -> Shut

Shut
    ^\! -> Record Start
    ^\s+${Shutdown} -> NoRecord Start

```

EOF

运行结果如下（将显示不带 shutdown 的接口信息）：

```

FSM Table:
['InterfaceName', 'IPAddress', 'Netmask', 'Shutdown']
['GigabitEthernet0/0', '10.1.1.1', '255.255.255.0', '']
['GigabitEthernet0/1', '172.16.1.1', '255.255.255.252', '']

```

## 5. 变量的属性

在本节第一个关于 LLDP 邻居的例子中，我们使用了一个变量属性“Filldown”。假设我们取消第一个例子中的变量属性，那么我们会得到如下结果：

```

FSM Template:
Value LocalHost (\S+)
Value LocalPort (\S+)
Value RemotePort (\S+)
Value RemoteHost (R\S+)
Value DeviceType ([R|B|T|C|W|P|S|O])

Start
    ^${LocalHost}[\#|>]show lldp
    ^${RemoteHost}\s+${LocalPort}\s+\d+\s+${DeviceType}\s+${RemotePort} -> Record

```

EOF

```

FSM Table:
['LocalHost', 'LocalPort', 'RemotePort', 'RemoteHost', 'DeviceType']
['R2', 'Gi0/2', 'Gi0/1', 'R4', 'R']
['', 'Gi0/1', 'Gi0/0', 'R3', 'R']
['', 'Gi0/0', 'Gi0/1', 'R1', 'R']

```

每次在匹配到一条新的记录时，所有变量的值都会被清空。这样，在后续的记录中这个变量就为空了。如果想保留这个值，我们可以使用 Filldown 属性。

除了 Filldown 属性之外，还有如下几个属性。

❑ Required：表示如果这个变量没有获取到值，而其他变量获取到了值，也将不记录

这条记录。

□ List: 表示这个变量是一个列表。可以有多个值在一条记录中。

关于这两个属性的使用, 我们可以参考 TextFSM 模块的文档 <https://github.com/google/textfsm/wiki/Code-Lab>。

### 11.1.5 在 Python 代码中使用 TextFSM

前面我们一直使用 TextFSM 模块的 textfsm.py 这个工具进行内容的格式化处理。除了这种方法, 还可以和其他模块一样, 在我们自己的代码中使用这个模块。下面是一个简单的例子。

```
#!/usr/bin/env python
# coding: utf-8
# 导入 TextFSM 模块
import textfsm

# 指定模板文件和需要处理的文本内容
TEMP_FILE = "interface_template"
INPUT_FILE = "interface.cfg"

# 打开模板文件, 并初始化 TextFSM 类
# 注意, 这里需要的是一个文件句柄, 而不是模板文件的内容
fsm = textfsm.TextFSM(open(TEMP_FILE))

# 读取需要处理的文本, 这里需要读取文件的内容, 而不是一个文件句柄
input_txt = open(INPUT_FILE).read()

# 使用 ParseText 方法解析文件
fsm_results = fsm.ParseText(input_txt)
# 输出变量名称
print(fsm.header)

# 逐行输出解析后的内容
for row in fsm_results:
    print(row)
```

为了比较清晰地说明问题, 我们在代码中添加了一些注释。其中, 加粗部分是主要的代码内容。其中文件“interface\_template”和“interface.cfg”的内容就是 11.1.4 节中的文本内容。读者可以自己运行一下这个代码, 这里就不在给出其运行结果了。

在本节中, 我们介绍了如何使用 TextFSM 来解析命令行的文本。利用好这个工具, 我们几乎可以处理所有命令行的输出和大部分网络设备的配置信息。使用这个工具, 我们可以得到一个结构化的数据, 这些结构化的数据方便后续进行进一步的处理和使用。关于 TextFSM 更多的帮助文件, 我们可以参考 <https://github.com/google/textfsm/wiki>。

## 11.2 Cisco 配置类型

正如本章一开始提到的，Cisco 大部分网络设备的配置都是通过行缩进的方法来区分配置块的。下面给出一个 IOS XR 设备 ISIS 的配置实例，这个配置分为五个配置块。其中，第一个配置块是 ISIS 的全局配置，另外四个配置块是 ISIS 的接口配置。很多时候，我们只需要获取某一部分的配置，比如我们希望获取接口类型是 point-to-point 的接口，又或者我们希望获取 metric 为 20 的接口信息。我们是否可以利用这种缩进格式来实现呢？

```
router isis t1
  is-type level-2-only
  net 49.0001.0000.0000.0001.00
  address-family ipv4 unicast
    metric-style wide
  !
  interface Loopback0
    passive
    address-family ipv4 unicast
      metric 10
    !
  !
  interface GigabitEthernet0/0/0/0
    point-to-point
    address-family ipv4 unicast
      metric 20
    !
  !
  interface GigabitEthernet0/0/0/1
    point-to-point
    address-family ipv4 unicast
      metric 30
    !
  !
  interface GigabitEthernet0/0/0/2
    address-family ipv4 unicast
      metric 40
    !
  !
!
```

我们可以使用另一个开源模块来处理这个需求。

### 11.2.1 ciscoconfparse 模块

从这个模块的名字“ciscoconfparse”就可以看出，它是为解析 Cisco 的配置文件而开发的，目前的版本是 1.2.49。其代码托管的位置为 <https://github.com/mpenning/ciscoconfparse>。目前这个模块除了能处理 Cisco 的配置文件外，还能处理以下这些厂家的配置文件：



- ❑ Arista EOS
- ❑ Brocade
- ❑ HP 交换机
- ❑ Force 10 交换机
- ❑ Dell 交换机
- ❑ Extreme
- ❑ Juniper JUNOS (1.2.4 以后)
- ❑ F5 (1.2.4 以后)

理解这个模块的功能首先需要了解这个模块对配置文件结构的定义。由于这些配置文件都是通过缩进的方式来区分配置块的，因此，在一个配置块中，“缩进少的行”定义为“缩进多的行”的父行，反之则是子行。配置如下：

```
interface GigabitEthernet0/0/0/0          # 父行
    point-to-point                        # 子行
    address-family ipv4 unicast           # 子行，又是 metric 20 的父行
        metric 20                        # 子行 (address-family 这一行的子行)
!
```

这种父与子的关系是理解这个模块的基础。目前这个模块不能直接处理孙行（子行的子行）的内容，这个需要通过 for 循环来处理，在后续的例子中我们将会详细介绍。

### 11.2.2 安装模块

在这之前，我们已经安装过许多模块了。这个模块同样可以通过 pip 来安装。命令如下：

```
$ pip install ciscoconfparse
Collecting ciscoconfparse
  Downloading ciscoconfparse-1.2.49.tar.gz (89kB)
    100% |=====| 92kB 948kB/s
Collecting ipaddress (from ciscoconfparse)
  Downloading ipaddress-1.0.18.tar.gz
Collecting dnspython3 (from ciscoconfparse)
  Downloading dnspython3-1.15.0.zip
Collecting colorama (from ciscoconfparse)
  Downloading colorama-0.3.9-py2.py3-none-any.whl
Collecting dnspython==1.15.0 (from dnspython3->ciscoconfparse)
  Downloading dnspython-1.15.0-py2.py3-none-any.whl (177kB)
    100% |=====| 184kB 2.0MB/s
Installing collected packages: ipaddress, dnspython, dnspython3, colorama, ciscoconfparse
Running setup.py install for ipaddress ... done
Running setup.py install for dnspython3 ... done
Running setup.py install for ciscoconfparse ... done
Successfully installed ciscoconfparse-1.2.49 colorama-0.3.9 dnspython-1.15.0
dnspython3-1.15.0 ipaddress-1.0.18
```

### 11.2.3 获取配置内容

在本节一开始我们给出了 Cisco IOS XR 平台上一个 ISIS 协议的配置，我们先来获取这个配置中接口为 point-to-point 类型的接口名称。其代码如下：

```
1 #!/usr/bin/env python
2 #coding:utf-8
3 from ciscoconfparse import CiscoConfParse
4
5 cfg = open("isis_ios.cfg").read().splitlines()
6
7 parse = CiscoConfParse(cfg)
8
9 for obj in parse.find_objects(r"interface"):
10     if obj.re_search_children(r"point-to-point"):
11         print(obj.text)
```

第 1~3 行，我们在之前的例子中已经解释过多次，这里不再赘述。

第 5 行，这行代码其实做了若干个操作。首先，打开一个名为 isis\_ios.cfg 的文件，在这里并没指定文件的路径，Python 解释器会在执行命令的目录下查找这个文件。默认参数下，open 函数会用只读的方式打开文件。其次，使用 read 方法读取文件中的内容。再次，使用 splitlines 方法对文本中的内容按照“行”分割并返回一个列表。最后，cfg 获得一个列表类型的变量。

第 7 行，使用模块提供的 CiscoConfParse 类进行初始化，即解析这个 cfg 列表类型的配置文件。

第 9~11 行，查找和输出相应的内容。首先，在第 9 行，我们查找包含“interface”字符的“行”对象，然后在这些对象的子行中查找包含“point-to-point”的信息。如果能找到，那么就输出这个接口的信息。

现在我们来运行一下这段代码：

```
$ python isisl.py
    interface GigabitEthernet0/0/0/0
    interface GigabitEthernet0/0/0/1
```

接下来，我们再来查找一下 metric 为 20 的接口。在 IOS XR 的配置中，metric 并不是接口的子行，而是孙行（子行的子行）。我们可以用下面的代码来实现：

```
1 #!/usr/bin/env python
2 #coding:utf-8
3 from ciscoconfparse import CiscoConfParse
4 cfg = open("isis_ios.cfg").read().splitlines()
5
6 parse = CiscoConfParse(cfg)
7
8 for obj in parse.find_objects(r"interface"):
9     af_v4 = obj.re_search_children(r"address-family ipv4")
```

```

10     for metric in af_v4:
11         if metric.re_search_children(r"metric 20"):
12             print(obj.text)

```

在这段代码中，我们只是先查找了 address-family ipv4，然后在 address-family ipv4 下面查找 metric 20 的信息。

运行结果如下：

```

$ python isis2.py
    interface GigabitEthernet0/0/0/0

```

### 11.2.4 修改设备配置

ciscoconfparse 模块除了能获取设备相应的配置部分，还能修改设备的配置。接下来，我们仍然基于本节开始的 ISIS 配置信息，我们希望对所有的以太网接口都添加 “point-to-point” 这个配置。我们可以用如下代码来实现。

```

1  #!/usr/bin/env python
2  #coding:utf-8
3  from ciscoconfparse import CiscoConfParse, IOSCfgLine
4  cfg = open("isis_ios.cfg").read().splitlines()
5
6  parse = CiscoConfParse(cfg)
7
8  Eth_re = "interface\s+\w+Ethernet"
9  for obj in parse.find_objects_wo_child(Eth_re, "point-to-point"):
10     obj.insert_after("point-to-point")
11
12  for line in parse.ioscfg:
13  print(line)

```

第 1~6 行，和之前的代码完全一样，这里不再赘述。

第 8 行，定义了以太网接口的正则表达式，用于查找接口。

第 9 行，使用了 find\_objects\_wo\_child 方法，其用于查找一个对象，是不是包含某个子行的内容。很显然，这里我们对已经包含 “point-to-point” 信息的接口不会再增加这条配置。

第 10 行，在配置中增加一行 “point-to-point” 配置，这个配置前有两个空格，这是因为在配置中增加两个空格更能体现它的层次结构。当然，在 IOS XR 中，不增加这个空格也是可以正常使用的。

第 12~13 行，输出修改后的全部配置。

运行结果如下：

```

router isis t1
    is-type level-2-only
    net 49.0001.0000.0000.0001.00
    address-family ipv4 unicast

```



```

metric-style wide
!
interface Loopback0
  passive
  address-family ipv4 unicast
    metric 10
  !
!
interface GigabitEthernet0/0/0/0
  point-to-point
  address-family ipv4 unicast
    metric 20
  !
!
interface GigabitEthernet0/0/0/1
  point-to-point
  address-family ipv4 unicast
    metric 30
  !
!
interface GigabitEthernet0/0/0/2
  point-to-point                                ! 新增加的一行配置
  address-family ipv4 unicast
    metric 40
  !
!
!

```

### 11.2.5 配置审计

对于本节刚开始提供的 ISIS 配置文件，如果我們希望在配置文件中检查所有的接口是否都包含了“point-to-point”以及“address-family ipv4 unicast”这两行配置。我们可以使用如下代码来实现：

```

1  #!/usr/bin/env python
2  #coding:utf-8
3  from ciscoconfparse import CiscoConfParse
4  cfg = open("isis_ios.cfg").read().splitlines()
5
6  parse = CiscoConfParse(cfg)
7
8  required_lines = [
9      "address-family ipv4 unicast",
10     "point-to-point",
11 ]
12
13 for obj in parse.find_objects(r"interface"):
14     p = CiscoConfParse(obj.ioscfg)
15     result = p.req_cfgspec_all_diff(required_lines)
16     if result:

```

```

17         print(obj.text)
18         print("missing config line(s):")
19         print("\n".join(result))
20         print("="*20)

```

第 9~12 行，定义一个列表，这个列表中的每一行都是需要进行检查的配置项。

第 14 行，查找接口部分的配置。

第 15 行，将一个接口的配置重新初始化为 CiscoConfParse 对象。

第 16 行，使用 req\_cfgspec\_all\_diff 方法进行配置的检查。

第 17~21 行，使用 if 语句来检查结果，如果发现有缺失的行，则输出相应的接口名称和缺失的内容。

运行结果如下：

```

$ python isis4.py
    interface Loopback0
missing config line(s):
point-to-point
=====
    interface GigabitEthernet0/0/0/2
missing config line(s):
point-to-point
=====

```

从这个结果，我们很容易看出哪些接口缺少了什么配置。

在这一节中，我们介绍了 ciscoconfparse 模块用于处理 Cisco 样式（通过空格进行缩进的格式）的配置文件。我们可以看到，这个模块不但可以用于查找配置文件中的内容，还可以用于修改配置文件。这样，我们就可以利用这个模块对 Cisco 样式的配置文件实现配置审计和配置生成等功能。

关于这个模块的更多功能，我们可以参考这个模块的文档：<http://www.pennington.net/py/ciscoconfparse/>。

## 11.3 JUNOS 配置类型

JUNOS 的配置文件和 Cisco IOS 的样式完全不一样。JUNOS 的配置文件有两种表现形式，一种是层次化配置形式，另一种是 set 命令格式配置形式。下面是这两种配置的例子。

层次化配置：

```

lab@rl# show protocols
isis {
    level 1 disable;
    level 2 wide-metrics-only;
    interface ge-0/0/0.0 {
        point-to-point;
        level 2 metric 10;
    }
}

```

```

    }
    interface ge-0/0/1.0 {
        point-to-point;
        level 2 metric 10;
    }
    interface ge-0/0/2.0 {
        level 2 metric 30;
    }
    interface lo0.0;
}

```

set 命令格式配置:

```

lab@r1# show protocols | display set
set protocols isis level 1 disable
set protocols isis level 2 wide-metrics-only
set protocols isis interface ge-0/0/0.0 point-to-point
set protocols isis interface ge-0/0/0.0 level 2 metric 10
set protocols isis interface ge-0/0/1.0 point-to-point
set protocols isis interface ge-0/0/1.0 level 2 metric 10
set protocols isis interface ge-0/0/2.0 level 2 metric 30
set protocols isis interface lo0.0

```

除了这两种配置形式之外, JUNOS 的配置文件还可以使用 XML 或者 JSON (14.1R2 后的版本) 格式来表示。但是, XML 和 JSON 的格式并不能直接在设备上配置, 其需要通过 netconf 或者 restconf 接口进行导入。在本节中, 我们只使用层次化配置或 set 命令格式配置这两种形式。

### 11.3.1 层次化配置

JUNOS 的层次化配置, 让人在阅读时能很快地理解其层次化的结构。虽然 JUNOS 可以使用 “{}” 来表示其层次的结构, 但是它和 JSON 等格式又有一些差异性, 笔者并没有发现比较成熟的 Python 模块来专门处理这样的格式。但是 11.2 节介绍的 ciscoconfparse 模块却可以用来处理, 其处理的思路是把分割符号 “{}” 和 “;” 去掉, 然后使用空格缩进来处理其文本内容。

```

1  #!/usr/bin/env python
2  #coding:utf-8
3  from ciscoconfparse import CiscoConfParse, IOSCfgLine
4  cfg = open("isis_junos.cfg").read().splitlines()
5
6  parse = CiscoConfParse(cfg, syntax='junos', comment='#!')
7
8  print("\n".join(parse.ioscfg))
9
10 print("Check interface without 'point-to-point' config line:")
11 for obj in parse.find_objects_wo_child("interface", "point-to-point"):
12 print(obj.text)

```



第 6 行，在初始化 `CiscoConfParse` 类时，指定了配置的语法格式为 “junos”，这样后面将会按照上述的思路来进行文本处理。

第 8 行，输出处理后的配置文件。在运行的结果中，我们可以查看这部分的内容。

第 10~12 行，后续的处理和之前的例子一样。这里查找的是接口中没有使用 “point-to-point” 的接口。

我们先运行一下这段代码，看看 `ciscoconfparse` 是如何处理 JUNOS 配置文件的。

```
1 $ python isis_juonsl.py
2 isis
3     level 1 disable
4     level 2 wide-metrics-only
5     interface ge-0/0/0.0
6         point-to-point
7         level 2 metric 10
8     interface ge-0/0/1.0
9         point-to-point
10        level 2 metric 10
11    interface ge-0/0/2.0
12        level 2 metric 30
13    interface lo0.0
14 Check interface without 'point-to-point' config line:
15     interface ge-0/0/2.0
16     interface lo0.0
```

第 2~13 行，输出去除 “{}” 和 “;” 之后的配置。删除这些区块分割符之后的配置就非常像 Cisco 空格缩进风格的配置了。

第 14~16 行，给出了查找的结果。

如果你只有 JUNOS 层次化的配置，那么这种处理的方法还是一个不错的选择。但是，这种方法不太方便用于配置的生成。在笔者看来，我们可以使用 JUNOS 中 `set` 命令的配置方式来完成相关功能。

### 11.3.2 set 命令行配置

对于熟悉 JUNOS 的读者而言，从设备上获取 `set` 命令行格式的配置输出是一件非常容易的事。而处理 `set` 命令风格的配置文件时，我们完全可以不用借助任何第三方模块，只用 Python 的基本数据类型就可以很好地处理。我们可以观察一下 `set` 命令行的配置，这种配置非常容易处理成一个二维的数组。

在本节开始我们给出了两个 JUNOS 的配置，后一个是 `set` 命令行的配置，这个配置与第一个层次化配置是完全等价的配置。下面我们处理的文件将基于 `set` 命令行的配置。

`set` 命令格式配置：

```
lab@r1# show protocols | display set
set protocols isis level 1 disable
```

```

set protocols isis level 2 wide-metrics-only
set protocols isis interface ge-0/0/0.0 point-to-point
set protocols isis interface ge-0/0/0.0 level 2 metric 10
set protocols isis interface ge-0/0/1.0 point-to-point
set protocols isis interface ge-0/0/1.0 level 2 metric 10
set protocols isis interface ge-0/0/2.0 level 2 metric 30
set protocols isis interface lo0.0

```

### 1. 在 set 中获取配置内容

和 11.2.3 节类似，我们将从 set 命令行的配置中获取那些接口中包含了“point-to-point”类型的接口名称。

代码如下：

```

1  #!/usr/bin/env python
2  #coding:utf-8
3
4  cfg = open("isis_set.cfg").read().splitlines()
5
6  for line in cfg:
7      if "point-to-point" in line:
8          line_list = line.split()
9          print(line_list[4])

```

在这段代码中，我们并没有导入任何模块，只是使用了 Python 字符串类型的几个基本方法就可以处理 set 命令行的配置。

第 4 行，我们在之前的例子中介绍过，目的是读取配置文件，并根据配置文件的行信息转换成一个列表。

第 6 行，用一个 for 循环遍历所有的行。

第 7 行，使用了 in 的方式进行字符串的匹配。如果需要完成正则表达式的匹配，我们可以使用 Python 自带的 re 模块进行查找。

第 8 行，对查找到的行使用 split 方法。split 方法默认会使用空格作为分割符把字符串转化为列表格式。

第 9 行，输出第 5 个元素。由于 Python 的列表类型的下标是从 0 开始的，第 5 个元素就是列表 4 元素。

运行结果如下：

```

$ python isis_junos_set.py
ge-0/0/0.0
ge-0/0/1.0

```

### 2. 修改配置文件

从上一个例子中，我们可以发现只有两个接口包含了“point-to-point”的配置。现在我们同样希望所有的以太口都添加这个配置，那么我们的代码可以通过以下方法来实现：

```

1  #!/usr/bin/env python

```

```

2 #coding:utf-8
3 import re
4 cfg = open("isis_set.cfg").read().splitlines()
5 interfaces = set()
6 p2p_infs = set()
7 interface_re = r"[x|g|e][e|t]\~\d+\/\d+\/\d+\/\d+"
8 for line in cfg:
9     line_split = line.split()
10    if len(line_split) > 5:
11        if re.match(interface_re, line_split[4]):
12            interfaces.add(line_split[4])
13            if "point-to-point" == line_split[5]:
14                p2p_infs.add(line_split[4])
15
16 no_p2p_infs = interfaces - p2p_infs
17 for inf in no_p2p_infs:
18     inf_cfg = "set protocols isis interface %s point-to-point" %inf
19     cfg.append(inf_cfg)
20
21 print("\n".join(cfg))

```

第5~6行，定义两个 set 类型的变量。在 Python 中，set 类型中的元素是不重复的元素。

第7行，定义一个 JUNOS 平台中以太口的正则表达式。在 JUNOS 中，GE 接口的表示以“ge-”开始，10GE 接口以“xe-”开始，40GE 和 100GE 接口以“et-”。

第8行，用 for 循环遍历配置中的所有行。

第9行，把一行的配置变成一个列表变量，这样方便后续的操作。

第10行，先判断在一行配置中里面的参数是否超过5个。如果不超过5个，当我们使用列表的下标5来获取第6个元素时，有可能存在列表操作失败的异常抛出。

第11行，使用正则表达式来匹配一行配置中的第5个元素，接口名称的信息会存在这里。建议对比着前面 set 命令格式配置进行查看和理解。

第12行，如果是接口名称，那么我们把这个接口名称添加到 set 类型的变量 interfaces 中。对于 set 类型，如果重复添加相同的元素将不会变化。最终，interfaces 变量里面将会存写所有以太口的接口名称信息。

第13~14行，判断一行配置中的第6个元素是不是“point-to-point”信息。如果是，那么就在 p2p\_infs 这个 set 类型的变量中添加接口名称。

第16行，两个 set 类型的变量相减，这里得到那些不是“point-to-point”接口类型的接口名称。

第17~19行，使用 append 方法增加一行配置到现有配置文件中。

第21行，输出所有更新后的配置信息。

大家也许注意到，新加的配置信息都被放到了配置文件的最后。在 JUNOS 中，使用 set 命令来添加配置是没有先后顺序的，因此这样做并不会带来什么问题。



### 3. 配置的审计

我们在前两个例子中介绍了如何获取和修改 JUNOS set 命令行的配置方法。审计 JUNOS set 命令行的配置文件也是非常的简单，这里笔者留给读者自己来思考如何撰写这样的代码。

## 11.4 小结

在本章中，我们主要介绍了两个模块。虽然这两个模块在某些功能上存在一些重叠，但它们还是各有各的特点。TextFSM 模块主要用于从半结构化的文本中获取一些数据，而 ciscoconfparse 模块主要用于 Cisco 样式配置文件的结构化，其结构化后将方便我们获取那些有上下文关联的配置信息。借助这两个模块，我们应该可以处理大部分网络设备命令行给出的信息以及网络设备自身的配置文件。

在日常工作，遇到过于复杂的信息，我们可以把这两个模块结合起来一起使用：先用 ciscoconfparse 对信息文本进行分块处理，然后用 TextFSM 来获取里面的数据。

获取网络设备给出的各项数据后，我们就需要来进一步处理这些数据。第 12 章将介绍网络环境中的两个比较特殊的数据形态，以及它们的处理方法。

## 网络特有数据类型处理

在第 11 章中，我们介绍了两个用于设备命令行输出处理和配置文件获取的模块。命令行输出和配置文件大多数是针对一些文本内容进行处理。对于网络而言，还有两种特有的数据类型是既需要经常处理，但又不方便通过文本处理方式解决的。这两种数据类型分别是**网络地址**和**网络拓扑**。

对于网络设备的功能而言，最核心的问题是处理（计算）好各自的路由信息；而对于网络的管理者而言，最核心的问题是管理好每台设备上的路由，从而实现整网的管理。路由最重要的部分由网络地址构成，最常见的网络地址包括 IP 地址（IPv4 和 IPv6）和 MAC 地址。Python 中处理 IP 地址的模块比较多，本章将重点介绍其中两个处理网络地址的模块。

进行网络设计与管理一定会用到网络拓扑，网络拓扑是网络工程师日常工作接触最多、最重要的内容之一。本章会介绍一个模块用来处理网络拓扑以及使用它来完成路径的计算。

### 12.1 Jupyter

下面我们先介绍一个 Python 编程工具 Jupyter。本章的后续内容都会使用这个工具来做代码演示。

Jupyter Notebook（此前被称为 IPython Notebook）是一个基于 Web 的交互式笔记本，目前可以支持 40 多种编程语言。在这里，我们将用它来编写交互式的 Python 代码，可以用它编写出非常漂亮的交互式笔记内容。

#### 12.1.1 安装 Jupyter

在开始使用 Jupyter Notebook 之前，我们需要先安装 Jupyter 工具。和其他 Python 模块

一样，我们可以通过 pip 进行安装。

```
$ pip install jupyter
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting jupyter-console (from jupyter)
  Downloading jupyter_console-5.2.0-py2.py3-none-any.whl
Collecting ipywidgets (from jupyter)
  Downloading ipywidgets-7.0.5-py2.py3-none-any.whl (68kB)
  100% |=====| 71kB 749kB/s
Collecting qtconsole (from jupyter)
  Downloading qtconsole-4.3.1-py2.py3-none-any.whl (108kB)
  100% |=====| 112kB 1.1MB/s
Requirement already satisfied: notebook in
<略>
  Downloading widgetsnbextension-3.0.8-py2.py3-none-any.whl (2.2MB)
  100% |=====| 2.2MB 627kB/s
Requirement already satisfied: traitlets>=4.3.1 in
<略>
Successfully installed ipywidgets-7.0.5 jupyter-1.0.0 jupyter-console-5.2.0
qtconsole-4.3.1 widgetsnbextension-3.0.8
```

## 12.1.2 启动 Jupyter

安装完成之后我们使用下面的命令启动 Jupyter。

```
$ jupyter notebook
```

运行完成后我们可以看到如下命令输出：

```
[I 10:40:49.295 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/?token=277307ab77a42d0ed9bb7df3202f9d607294bfdc54ec7fa0
[I 10:40:49.295 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 10:40:49.296 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token: http://localhost:8888/?token=277307ab77a42d0ed9bb7df3202f9d607294bfdc54ec7fa0
[I 10:40:49.582 NotebookApp] Accepting one-time-token-authenticated connection from ::1
```

这时系统会打开默认浏览器，接着我们就可以在浏览器中使用 Jupyter Notebook。如果系统并没有正常打开，可以复制、粘贴加粗部分的输出到浏览器里面直接打开和使用。图 12-1 是 Jupyter 运行的界面。

默认情况下，Jupyter 会启动一个只能本地访问的 Web 服务，端口号默认为 8888。如果修改其默认参数，可以实现基于 IP 地址和服务端口的访问。另外考虑到安全问题，还可以给 web 登录设置一个登录密码。操作的具体步骤如下。

首先，生成一个默认的配置文件。



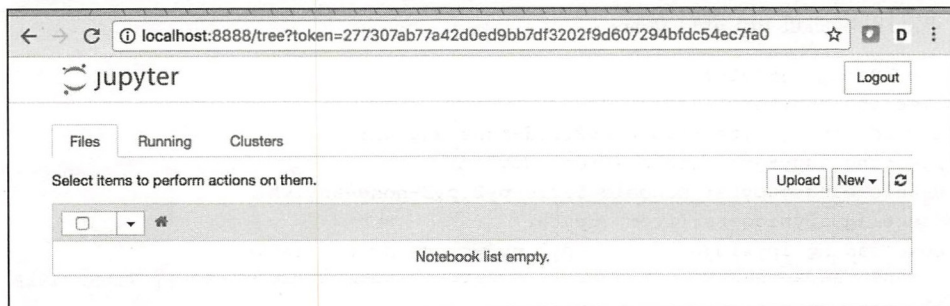


图 12-1 Jupyter 运行界面

```
$ jupyter notebook --generate-config
Writing default config to: /Users/xinyu3/.jupyter/jupyter_notebook_config.py
```

这里加粗部分是默认配置文件所保存的位置和名称。

然后，我们使用下面的方法来生成登录密码，其方式为 SHA1 散列 (HASH) 密码。

```
$ python
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from notebook.auth import passwd; passwd("lab123")
'sha1:daea3930704b:c4fd39be5c0a7be0cd2980ea42182f5cd0f163ae'
```

最后，打开上面的配置文件修改两个参数，设置 IP 地址和登录密码。

```
c.NotebookApp.ip = "172.17.11.19"
c.NotebookApp.password = 'sha1:daea3930704b:c4fd39be5c0a7be0cd2980ea42182f5cd0f163ae'
```

我们再次运行 Jupyter 服务：

```
$ jupyter notebook
< 略 >
[I 13:07:45.517 NotebookApp] 0 active kernels
[I 13:07:45.517 NotebookApp] The Jupyter Notebook is running at: http://172.17.11.19:8888/
[I 13:07:45.517 NotebookApp] Use Control-C to stop this server and shut down all kernels
(twice to skip confirmation).
```

这里我们就可以看到图 12-2 所示的 Jupyter 登录界面。

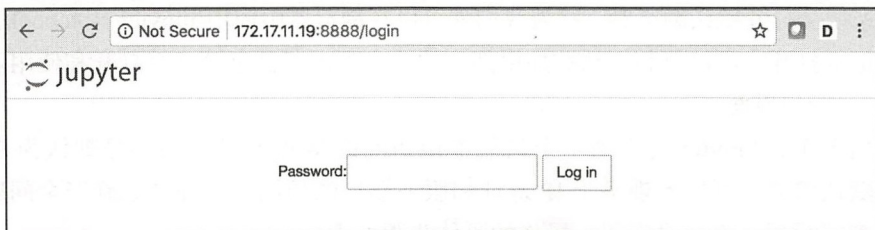


图 12-2 Jupyter 登录界面

### 12.1.3 使用 Jupyter

登录成功后，我们可以看到页面的右上角有一个新建按钮。这里我们可以新建一个 Python 3 Notebook 文件，如图 12-3 所示。

在新打开的页面中，我们会看到 Notebook 界面，其实此时页面里面什么内容也没有，如图 12-4 所示。

现在我们就可以在这里写入代码，每一个代码单元格（code cell）都会立刻给出返回结果。我们接下来测试两个简单的代码，如图 12-5 所示。

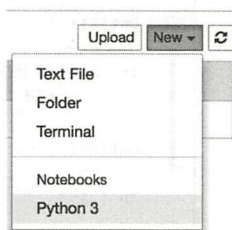


图 12-3 新建 Python 3 Notebook

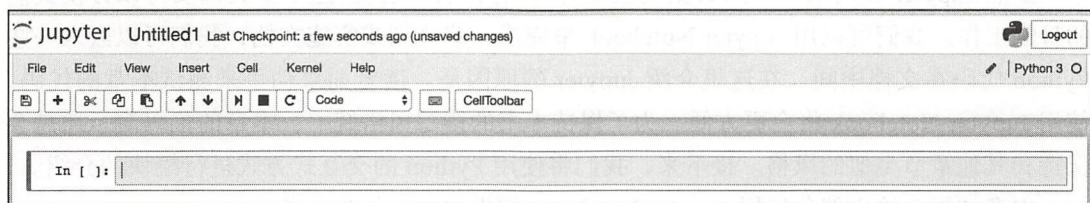


图 12-4 Jupyter Notebook 界面

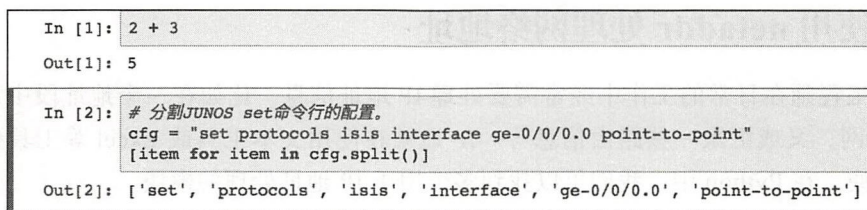


图 12-5 Jupyter Notebook 代码

每一个代码单元格（code cell）的输入文本框左边以 “In [ ]:” 开头。在这种类型的单元格中，可以输入任意 Python 代码并执行。例如，我们输入  $2+3$  并按下 Shift+Enter 组合键。单元格中的代码就会被立刻执行，光标也会移动到一个新的代码单元格中。在新的代码单元格的上面会有上一个代码单元格的运行结果。

Jupyter Notebook 有一个不错的功能：我们可以单独运行具体某一个代码单元格，而不影响其他单元格。这个特性为我们测试代码带来了很大的方便，我们修改了第一个单元格，并重新运行了一次，结果如图 12-6 所示，可以看到第一个代码单元格的结果相应地修改了，而第二个代码单元格没有任何变化。

此外，Jupyter Notebook 中可以增加基于 Markdown 标记的文本内容，这给代码提供了非常丰富的解释能力。我们还可以把 Notebook 导出为 HTML 或者 PDF 格式的文件。

更多关于 Jupyter Notebook 的内容，我们可以参考 <http://jupyter-notebook.readthedocs.io>。

```
In [3]: 2 + 3 + 4
Out[3]: 9

In [2]: # 分割JUNOS set命令行的配置。
        cfg = "set protocols isis interface ge-0/0/0.0 point-to-point"
        [item for item in cfg.split()]

Out[2]: ['set', 'protocols', 'isis', 'interface', 'ge-0/0/0.0', 'point-to-point']

In [ ]:
```

图 12-6 再次运行第一个单元格

虽然 Jupyter 和网络特有的数据类型没有任何关系，但借助这个工具确实可以方便我们的日常工作。我们可以用 Jupyter Notebook 来完成一些规划或实施文档，我们可以嵌入一些 Python 代码在文档中间。在这里介绍 Jupyter 的原因是：读者如果动手来测试本章的代码，使用 Jupyter Notebook 将会更方便。为了保持本书的撰写风格统一，本章的代码演示还是会保持和其他章节类似的风格。接下来，我们将使用 Python 的交互式方式进行模块的介绍。

本章结尾会给出部分使用 Jupyter Notebook 编辑的文件作为参考。

## 12.2 使用 netaddr 处理网络地址

网络工程师在日常的工作中经常需要处理 IP 地址信息，比如在一个地址段中重新分配 IP 地址子网，又或汇聚一些路由信息等。IP 地址在使用文本工具或 Excel 等工具处理时并不是很方便。在 Python 中，我们可以找到多个用于 IP 地址处理的模块。

### 12.2.1 安装 netaddr 模块

通过前几章的介绍，我们应该熟悉了 Python 模块的安装方法。netaddr 模块可以用 pip 进行安装。

```
$ pip install netaddr
```

### 12.2.2 IP 地址的基本属性

netaddr 模块有两个基本类，分别是 IPAddress 和 IPNetwork。这两个名词对于网络工程师而言应该不会产生什么歧义。我们在处理 IP 地址时，大部分情形下会围绕着这两个类进行。

假设有一个 IP 地址为“192.168.7.83/27”，对于这样的 IP 地址形式，想必大家并不陌生。我们在日常工作中，经常需要获取这个 IP 地址的相关信息，比如它的子网掩码是什么，它的广播地址是什么，它的网络地址（路由前缀）是什么，又或者它的反掩码（常用于 ACL 中）是什么，等等。下面我们将用 netaddr 模块来获取以上这些信息。

我们可以直接在 Linux、MAC OSX 或 Windows（已经正常安装了 Python）的命令行中



输入 python（或 python3）进入 Python 的交互模式，这里以 MAC OSX 操作为例。

```
$ python
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

这里我们将获得“>>>”提示符，提示符中包含一个空格。

我们先导入 netaddr 模块中的 IPAddress 类和 IPNetwork 类。

```
>>> from netaddr import IPAddress, IPNetwork
>>>
```

然后使用 IPNetwork 初始化（注意：由于在 Python 中空格缩进有语法作用，因此不要使用空格开头）。

```
>>> ip = IPNetwork("192.168.7.83/27")
```

1) 获取子网掩码。

```
>>> ip.netmask
IPAddress('255.255.255.224')
```

2) 获取广播地址。

```
>>> ip.broadcast
IPAddress('192.168.7.95')
```

3) 获取网络地址。

```
>>> ip.network
IPAddress('192.168.7.64')
```

4) 获取反掩码。

```
>>> ip.hostmask
IPAddress('0.0.0.31')
```

5) 获取所在网段一共包含多少个 IP 地址。

```
>>> ip.size
32
```

6) 修改地址的掩码长度为 28。

```
>>> ip.prefixlen = 28
>>> ip
IPNetwork('192.168.7.83/28')
```

### 12.2.3 处理 IP 地址的基本方法

1) 把此 IP 地址所在的网段分为掩码长度为 30 的几个子网。

```
>>> [x for x in ip.subnet(30)]
```

```
[IPNetwork('192.168.7.80/30'), IPNetwork('192.168.7.84/30'), IPNetwork('192.168.7.88/30'),  
IPNetwork('192.168.7.92/30')]
```



注意 上面的 IP 已经被赋值为 192.168.7.83/28。

2) 获取某一段地址中的所有主机地址。

```
>>> for ip in IPNetwork("172.16.10.34/29").iter_hosts():  
... print(ip)  
...  
172.16.10.33  
172.16.10.34  
172.16.10.35  
172.16.10.36  
172.16.10.37  
172.16.10.38
```

这个方法只获取主机地址，已经排除了这个地址所在网段的网络地址和广播地址。

3) 判断地址的公网与私网属性。

```
>>> IPAddress("100.64.0.1").is_private()  
True  
>>> IPAddress("114.114.114.114").is_private()  
False
```

100.64.0.0/10 是 RFC 6598 中预留的私网地址段。

4) 定义任意一段 IP 地址范围。

```
>>> from netaddr import IPRange  
>>> ip_range = IPRange("192.168.1.10", "192.168.1.130")
```

5) 对这一段地址范围进行地址聚合。

```
>>> ip_range.cidrs()  
[IPNetwork('192.168.1.10/31'), IPNetwork('192.168.1.12/30'), IPNetwork('192.168.1.16/28'),  
IPNetwork('192.168.1.32/27'), IPNetwork('192.168.1.64/26'), IPNetwork('192.168.1.  
128/31'), IPNetwork('192.168.1.130/32')]  
>>>
```



注意 这里是 IPRange 范围内所有 IP 地址的精确聚合。

## 12.2.4 IP 地址的加减法

IPAddress 类型的地址可以直接和一个整数进行加减法运算，这样的运算可以方便我们进行地址分配。

```
>>> ip = IPAddress("172.20.1.1")  
>>> ip + 256  
IPAddress('172.20.2.1')
```

```
>>> ip - 3
IPAddress('172.20.0.254')
```

这里我们用 IP 地址的加法来完成对接口地址的分配并生成配置。假设我们要基于以下接口配置模板来生成 10 个接口的相应配置：

```
interface GigabitEthernet0/0/0/1.2
    ipv4 address 172.20.1.1 255.255.255.252
    encapsulation dot1q 2
!
```

在这个配置中，所有的加粗部分都是需要变化的配置。

这里的代码如下：

```
1 #!/usr/bin/env python
2 from netaddr import IPNetwork
3
4 inf_cfg = '''
5 interface GigabitEthernet0/0/0/1.%d
6     ipv4 address %s 255.255.255.252
7     encapsulation dot1q %d
8 !
9 '''
10
11 ip_net = IPNetwork("172.20.1.1/24")
12 for i in range(1, 11):
13     ip = ip_net.ip + (i - 1) * 4
14     print(inf_cfg %(i, ip, i))
```

第 2 行，导入 netaddr 模块中的 IPNetwork 类。

第 4~8 行，定义一个接口配置的模板。

第 11 行，定义一个变量 ip\_net，这个变量为 IP 网段的第一个地址。

第 12~14 行，使用 for 循环输出 10 个接口的配置。

运行结果如下：

```
$ python interface_cfg.py

interface GigabitEthernet0/0/0/1.1
    ipv4 address 172.20.1.1 255.255.255.252
    encapsulation dot1q 1
!
< 略 >
interface GigabitEthernet0/0/0/1.10
    ipv4 address 172.20.1.37 255.255.255.252
    encapsulation dot1q 10
!
```

### 12.2.5 地址的聚合

刚才我们在 IPRange 类的方法中完成了一次地址的聚合操作。除了这种方法外，还可



以采用其他方法来完成地址的聚合功能。例如：

```
>>> ip_list = []
>>> ip_list.append(IPNetwork("192.168.1.0/25"))
>>> ip_list.append(IPNetwork("192.168.1.128/25"))
>>> ip_list.append(IPNetwork("192.168.0.0/24"))
>>> ip_list.append(IPNetwork("192.168.3.0/26"))
>>> ip_list.append(IPNetwork("192.168.3.64/26"))
>>> cidr_merge(ip_list)
[IPNetwork('192.168.0.0/23'), IPNetwork('192.168.3.0/25')]
```

这个方法可以方便地用于路由汇总的计算。

## 12.2.6 IPv6 地址

这里的所有例子都是基于 IPv4 的，我们并没有给出 IPv6 的例子。其实 netaddr 对 IPv6 的支持也非常完善，读者可以根据上面的例子把 IPv4 地址替换为 IPv6 地址，然后进行一些尝试。其文档可以参考 <https://netaddr.readthedocs.io/en/latest>。

## 12.2.7 使用 netaddr 处理 MAC 地址

netaddr 模块不仅可以处理 IP 地址信息，还可以处理 MAC 地址。MAC 地址并不像 IP 一样有很多丰富的属性，但 MAC 地址的主要问题是其表示方式的多样性。另外，通过 MAC 地址，我们通常希望知道它属于哪个厂家。

```
>>> from netaddr import *
>>> mac = EUI("98:5a:eb:9f:35:f8")
>>> mac
EUI('98-5A-EB-9F-35-F8')
```

在初始化 MAC 地址的时候，MAC 地址可以使用多种不同的格式。下面给出几种常见格式。

```
>>> EUI("98-5a-eb-9f-35-f8")
EUI('98-5A-EB-9F-35-F8')

>>> EUI("985a-eb9f-35f8")
EUI('98-5A-EB-9F-35-F8')

>>> EUI("985a:eb9f:35f8")
EUI('98-5A-EB-9F-35-F8')
```

MAC 地址的显示格式也支持多种方式输出。

```
>>> mac.dialect = mac_unix
>>> mac
EUI('98:5a:eb:9f:35:f8')

>>> mac.dialect = mac_cisco
```

```
>>> mac
EUI('985a.eb9f.35f8')
>>> mac.dialect = mac_bare
>>> mac
EUI('985AEB9F35F8')
```

我们还可以获取 MAC 地址大部分厂家的信息。

```
>>> mac.info
{'OUI': {'address': ['1 Infinite Loop', 'Cupertino CA 95014', 'UNITED STATES'],
        'idx': 9984747,
        'offset': 2953912,
        'org': 'Apple, Inc.',
        'oui': '98-5A-EB',
        'size': 132}}
```

通过上面的这些例子，我们可以看到使用 `netaddr` 模块处理网络地址可以带来很大的便捷性，甚至我们可以使用这个模块来模拟路由器路由查找的功能，当然，简单的模拟并不能实现路由器自身路由查找的高效性。

## 12.3 使用 `ipaddr` 处理网络地址

在 12.2.7 节中，我们使用 `netaddr` 模块处理了 IP 地址和 MAC 地址的数据类型。下面我们将介绍一个类似的模块用于 IP 地址数据的处理。

### 1. `ipaddr` 模块简介

这是一个由 Google 开源的小项目，其代码托管在 <https://github.com/google/ipaddr-py>。这个模块和 `netaddr` 一样，都能处理 IPv4 和 IPv6 地址类型，但是它并不提供 MAC 地址的处理能力。从 Python 3.3 起，这个模块已经被加入到 Python 的标准库中，其模块名修改为 `ipaddress`。在加入到标准模块后，此模块也做了一些小的修改，因此 `ipaddress` 模块和 `ipaddr` 模块在部分细节上还是存在着一些差异。

`ipaddr` 模块的所有功能都在一个文件中实现，它是一个轻量级的 IP 处理模块。

### 2. 安装 `ipaddr` 模块

对于 Python 3.3 以上的版本，可以直接使用 `ipaddress` 模块。如果你还是希望使用原来的 `ipaddr` 模块或者当前使用的 Python 版本低于 3.3，那么和其他模块的安装方法一样，可以通过 `pip` 安装 `ipaddr` 模块。

```
$ pip install ipaddr
```

### 3. `ipaddr` 模块的基本使用方法

这里我们以 `ipaddr` 为例，介绍 `ipaddr` 的基本属性和部分使用方法。这里，我们仍然以在 MAC OSX 中使用 Python 的交互形式进行演示。

首先，在命令行中输入 python（或者是 python3）进入 Python 的交互式界面。

```
$ python
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

其次，导入 ipaddr 模块中的所有方法。在实际的代码编写过程中，并不推荐使用这种形式来导入模块，因为这样做你并不清楚在代码中导入了哪些方法或类。尤其是当有多个模块都被导入时，在代码中可能会存在方法或类名称的重复，从而导致代码执行时出现异常。

```
>>> from ipaddr import *
>>>
```

最后，初始化两个 IPNetwork 对象，一个是 IPv4 地址，另一个是 IPv6 地址。我们看到这两个地址的初始化方法是完全一样的（在 netaddr 模块中也可以这么做）。

```
>>> ipv4 = IPNetwork("192.168.33.83/28")
>>> ipv6 = IPNetwork("2017:11:AF::17/56")
```

#### （1）获取子网掩码

```
>>> ipv4.netmask
IPv4Address('255.255.255.240')
>>> ipv6.netmask
IPv6Address('ffff:ffff:ffff:ff00::')
>>>
```

通常我们很少使用 IPv6 的 netmask。对于 IPv6 地址，我们一般使用前缀的长度。在初始化 IPNetwork 的时候，我们既可以使用前缀长度，也可以使用子网掩码。

```
>>> ipv4_1 = IPNetwork("192.168.34.92/255.255.255.248")
>>> ipv4_1
IPv4Network('192.168.34.92/29')
```

#### （2）获取广播地址

```
>>> ipv4.broadcast
IPv4Address('192.168.33.95')
```

#### （3）获取网络地址

```
>>> ipv4.network
IPv4Address('192.168.33.80')
```

#### （4）获取反掩码

```
>>> ipv4.hostmask
IPv4Address('0.0.0.15')
```



### (5) 获取地址的包含关系

```
>>> IPAddress("192.168.33.84") in ipv4
True
```

### (6) 地址聚合

```
>>> ip_list = []
>>> ip_list.append(IPNetwork("1.1.1.0/25"))
>>> ip_list.append(IPNetwork("1.1.1.128/25"))
>>> ip_list.append(IPNetwork("1.1.0.0/24"))
>>> ip_list.append(IPNetwork("1.1.2.0/23"))
>>> CollapseAddrList(ip_list)
[IPv4Network('1.1.0.0/22')]
```

我们可以看到 `ipaddr` 的很多属性或方法和 `netaddr` 模块还是很类似的。这两个模块在 IP 地址处理的功能方面比较接近，读者可以自行任意选择。更多关于 `ipaddr` 的属性和方法请参考 <http://pythonhosted.org/ipaddr/>。

## 12.4 网络拓扑的处理

网络拓扑是网络工程师日常工作的基础。我们不论是在网络规划阶段、网络建设阶段还是在维护阶段都离不开网络拓扑。通常我们会使用一些画图工具来完成网络拓扑的绘制，比如用 Microsoft Visio 或 Office PowerPoint 来绘制网络拓扑图。使用这样的工具可以画出非常漂亮的拓扑图，但是画出来的拓扑图并不方便转化为结构化的数据格式，我们也很难在这样的拓扑图上完成与拓扑相关的路径计算。

我们在第 2 章中提到过 DOT 语言，这是一种用于描述网络拓扑的语言（拓扑数据结构的描述性语言）。本节将进一步介绍 DOT 语言。另外，我们还将介绍一个用于计算网络拓扑相关内容的 Python 模块。

### 12.4.1 描述一个网络拓扑

在数学的概念中，“图论”研究的是顶点和边所组成的图形。计算机网络拓扑是数学概念“图”的一个子集，因此计算机网络拓扑图也可以由节点（即顶点）和链路（即边）来进行定义和绘制。根据不同的划分维度，我们可以将网络拓扑图进行如下划分。

#### 1. 无向图

在较为简单的场景中，我们通常用无向图来表示网络拓扑。在这样的拓扑图中，节点和节点之间的连接是没有方向的。这是因为在计算机网络中，通常链路都是双向的且都能进行数据传递，此时我们并不太关注（其实也无须关注）链路的方向。这种图通常在物理连接拓扑图中最为常用。我们可以用 DOT 语言来表示：



```
graph site_a {
    core1 -- access1;
    core2 -- access2;
    core1 -- core2;
}
```

首先，我们使用关键字 `graph` 作为一个无向图的开始，后面 `site_a` 是这个无向图的名称。然后，使用大括号 “{}” 来描述所包含的节点。节点和节点之间的关系（边）使用 “--” 双连字号来表示。每一行关系描述的末尾使用分号 “;”。对于上面 DOT 语言描绘的拓扑图，我们可以用 `chrome` 浏览器加插件来查看，如图 12-7 所示。

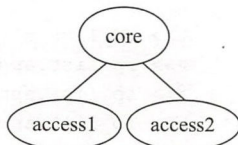


图 12-7 无向图

## 2. 有向图

计算机网络中流动的是数据流，而数据流其实有很强的方向性。当需要在拓扑图中加入方向时，我们就可以用有向图来表示。定义有向图的关键字为 `digraph`，使用箭头来表示链路（节点与节点之间的互连关系）。代码如下：

```
digraph site_b {
    core1 [shape=box];
    core2 [shape=box];
    access1;
    access2;
    access1 -> core1 -> core2 -> access2;
}
```

我们可以在节点或边后面添加一些属性。上面的例子定义了 `core1` 和 `core2` 的形状为方形。

同样，我们可以在 `chrome` 中查看这个代码对应的拓扑图，如图 12-8 所示。

## 3. 多重图

多重边图（multi-grahp）也称为伪图（pseudograph），这是允许存在多重边的图，也就是两个节点之间存在多条边（链路）。

众所周知，计算机网络中节点与节点之间经常存在多条链路。有时，这些链路的属性还是不一样的，比如两个节点之间既存在 10GE 链路，又存在 100GE 链路。

这时，我们在 DOT 代码中只需要多次添加边就可以了。例如：

```
graph site_c {
    core1 -- core2 [label="10GE*8",color="red",fontsize=9.0];
    core1 -- core2 [label="100GE*2",color="blue",fontsize=9.0];
}
```

这里，我们在节点 `core1` 和 `core2` 之间添加了两条边，通过为边添加属性，我们可以知

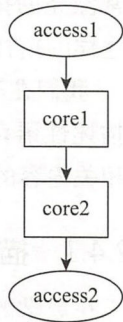


图 12-8 有向图



道第一条边代表八个 10GE 的链路，第二条边代表两个 100GE 链路；并且分别为这两条链路设置了不同的颜色。

现在，我们在浏览器中可以看到图 12-9 所示的拓扑。

更多关于 DOT 语言的描述请参考 <https://graphviz.gitlab.io/document->

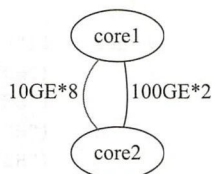


图 12-9 多重图

ation。  
本节介绍了如何用 DOT 语言来表示（绘制）一个网络拓扑。在接下来的两节中，我们将介绍如何用 networkx 这个模块对拓扑进行路径的计算。

## 12.4.2 最短路径的计算

在 IP 网络中，IGP 常用的有 OSPF 和 ISIS 两种协议，如何使用这两种协议对于网络工程师而言应该不会陌生。这两个协议的一个重要共同点是：它们都是基于链路状态的路由协议。基于链路状态的路由协议在进行路由表计算时均采用 SPF（Shortest Path First，最短路径优先）算法。在日常的工作中，网络工程师通常不需要自己使用 SPF 算法来计算路由器上的路由表条目，因为网络设备都会准确无误地计算其自身的路由信息。但我们在做网络规划和网络运维时经常需要对网络进行路径分析，这就需要我们做大量的拓扑仿真计算，这些计算大部分是基于 SPF 算法的。

在 Python 开源模块中，networkx 是一个功能丰富的用于拓扑计算的模块。现在的软件版本为 2.0，后续内容的介绍将会基于这个版本。和其他模块的介绍一样，我们还是从模块的安装开始。

对于 Python 模块，我们还是推荐使用 pip 进行软件的安装。

```

$ pip install networkx
Collecting networkx
Using cached networkx-2.0.zip
Collecting decorator>=4.1.0 (from networkx)
Using cached decorator-4.1.2-py2.py3-none-any.whl
Installing collected packages: decorator, networkx
Running setup.py install for networkx ... done
Successfully installed decorator-4.1.2 networkx-2.0
  
```

### 1. 用 networkx 描述拓扑

为了计算网络的路径，我们需要先输入网络的拓扑关系，有了网络的拓扑关系才可以进行基于路径拓扑的计算。在网络拓扑的描述上，networkx 和 DOT 语言非常类似，也分为三个部分：节点、边以及属性。在 networkx 中，节点被称为 node，边是链路，被称为 edge。

```

#!/usr/bin/env python
#coding:utf-8
import networkx as nx

nodes = ["BJ", "SH", "GZ", "HZ", "NJ", "WH", "XA"]
G = nx.Graph()
  
```





```

for node in nodes:
    G.add_node(node)

edges = [("BJ", "SH"),
         ("BJ", "GZ"),
         ("SH", "GZ"),
         ("HZ", "SH"),
         ("HZ", "GZ"),
         ("NJ", "SH"),
         ("NJ", "BJ"),
         ("WH", "SH"),
         ("WH", "BJ"),
         ("XA", "GZ"),
         ("XA", "BJ"),]
G.add_edges_from(edges)

```

上面的代码只是添加了一个拓扑。和 DOT 一样，networkx 对拓扑也分为有向图和无向图、单重和多重图。上面的例子用 `nx.Graph()` 初始化了一个无向的单重图。如果要创建有向图，则需要使用 `nx.DiGraph()`，创建多重图使用 `nx.MultiGraph()`，创建有向多重图使用 `nx.MultiDiGraph()`。

后续的代码使用了两个方法来添加节点（node）和边（edge）。

这个拓扑图可以参考图 12-10。

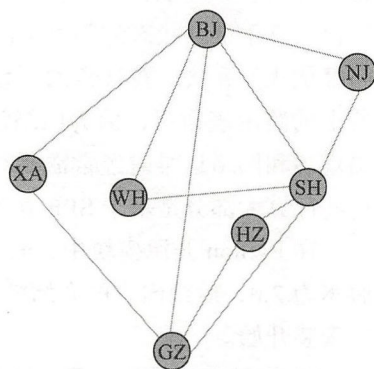


图 12-10 拓扑图示意图

## 2. 最简单路径计算

在 OSPF 和 ISIS 协议中，我们通常基于链路的 cost 来计算最短路径。在上面的例子中，我们可以假设所有链路的 cost 值都为 1。我们先计算一下节点 WH 和 HZ 之间的最短路径，需要用到 networkx 自带的算法来进行计算。这里我们使用 `shortest_path` 方法来计算两个节点之间的最短路径。

```

>>> nx.shortest_path(G, "WH", "HZ")
['WH', 'SH', 'HZ']

```



这里使用了 Python 的交互模式，这段代码的背景就是上一个例子中的代码内容。

## 3. 给链路添加 cost

在拓扑定义的时候可以添加 cost，networkx 默认使用 weight 作为 cost 属性。

```

#!/usr/bin/env python
#coding:utf-8
import networkx as nx

```



```
nodes = ["BJ", "SH", "GZ", "HZ", "NJ", "WH", "XA"]
G = nx.Graph()
for node in nodes:
    G.add_node(node)
```

```
edges = [("BJ", "SH", 1200),
          ("BJ", "GZ", 2500),
          ("SH", "GZ", 1300),
          ("HZ", "SH", 280),
          ("HZ", "GZ", 1000),
          ("NJ", "SH", 300),
          ("NJ", "BJ", 900),
          ("WH", "SH", 800),
          ("WH", "BJ", 850),
          ("XA", "GZ", 2600),
          ("XA", "BJ", 2000),]
```

```
G.add_weighted_edges_from(edges)
```

这个拓扑的连接关系和上一个例子是一样的，只是在连接关系上添加了 cost 值（其值是笔者基于距离的值，不过这个距离并不是严格依照地图的距离，而是做了一些简单修改后的值）。

现在我们再用刚才的算法计算一下 XA 到 HZ 的最短路径。

```
>>> nx.shortest_path(G, "XA", "HZ")
['XA', 'GZ', 'HZ']
```

在默认情况下，方法 `shortest_path` 并不会考虑 cost 值的大小情况，还是基于跳数来计算。我们可以看到 XA 到 HZ 是经过 GZ 节点的。

我们修改 `shortest_path` 的参数：

```
>>> nx.shortest_path(G, "XA", "HZ", weight="weight")
['XA', 'BJ', 'SH', 'HZ']
```

修改参数后 `shortest_path` 使用了 `weight` 值作为 cost 进行计算，计算的最短路径是 `weight` 值的和最小的路径。

除了一开始设置的链路 cost 值之外，还可以单独修改某条链路的 cost 值。比如现在假设 HZ 和 SH 之间的链路发生了故障，我们可以修改这条链路的 cost 为一个很大的值，假设为 1000000。然后再计算一次 XA 到 HZ 的最短路径。

```
>>> G["HZ"]["SH"]["weight"] = 1000000
>>> nx.shortest_path(G, "XA", "HZ", weight="weight")
['XA', 'GZ', 'HZ']
```

经过修改后，XA 到 HZ 的最短路径发生了变化，不再经过 BJ 和 SH 节点。

#### 4. 等价路径的计算

实际环境中，我们的网络中存在大量的等价路径，这时查找出所有的等价路径是很有必要的。在上面的这个拓扑中，如何查找出 WH 到 GZ 的所有等价路径呢？方法如下：



```
>>> list(nx.all_shortest_paths(G, "WH", "GZ", weight=None))
[['WH', 'BJ', 'GZ'], ['WH', 'SH', 'GZ']]
```

这里我们用了一个新的方法 `all_shortest_paths`，这个方法会返回一个生成器对象。我们可以用 `list()` 来遍历出所有的值。在上面的例子中，我们可以找到两条等价路径（并没有考虑链路 `cost` 值，只考虑了跳数）。

`networkx` 还提供了很多关于最短路径的方法，读者可以参考 [https://networkx.github.io/documentation/stable/reference/algorithms/shortest\\_paths.html](https://networkx.github.io/documentation/stable/reference/algorithms/shortest_paths.html)。

### 12.4.3 可用路径的计算

由于 IP 网络的 IGP 大多是基于最短路径的，因此在大部分情况下，我们只需要和 IGP 一样计算出最短路径就可以了。毕竟基于最短路径的流量模型是最为经济的，在大多数情况下，通过动态修改链路的 `cost` 值可以达到最优的流量分配。

当 IP 网络有了流量工程之类的工具，无论是通过 RSVP-TE 还是 Segment Routing TE (SRTE) 来部署流量的转发路径，在这之前通常都需要先获得流量的可用路径。而这些可用路径除了最短路径以外，还有那些全程路径 `cost` 值相对大一些的路径，我们可以称这些路径为次优路径或次次优路径。`networkx` 中也有这样的算法供大家使用。

#### 1. 获得可用路径

首先，可用路径是那些不存在节点重复的路径。假设我们有 A、B、C 三个节点，这三个节点形成了一个三角形拓扑，见图 12-11。在这个拓扑中，A 和 C 之间的可用路径有两条：一条是 A 到 C（我们这里记录为 A → C），另一条是 A 到 B 再到 C（记录为 A → B → C）。这里不存在第三条可用路径，路径 A → B → A → C 或 A → B → A → B → C 这样的路径并不是我们这里所说的可用路径，因为在这两条路径中，对于同一个节点出现了重复。

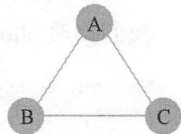


图 12-11 三角形拓扑

现在我们还是基于 12.4.2 节中的拓扑（见图 12-10）来获取 XA 和 HZ 之间的所有可用路径。

```
>>> for path in nx.all_simple_paths(G, "XA", "HZ"):
...     print(path)
...
['XA', 'BJ', 'NJ', 'SH', 'GZ', 'HZ']
['XA', 'BJ', 'NJ', 'SH', 'HZ']
['XA', 'BJ', 'WH', 'SH', 'GZ', 'HZ']
['XA', 'BJ', 'WH', 'SH', 'HZ']
['XA', 'BJ', 'GZ', 'HZ']
['XA', 'BJ', 'GZ', 'SH', 'HZ']
['XA', 'BJ', 'SH', 'GZ', 'HZ']
['XA', 'BJ', 'SH', 'HZ']
['XA', 'GZ', 'HZ']
['XA', 'GZ', 'BJ', 'NJ', 'SH', 'HZ']
['XA', 'GZ', 'BJ', 'WH', 'SH', 'HZ']
```





```
['XA', 'GZ', 'BJ', 'SH', 'HZ']
['XA', 'GZ', 'SH', 'HZ']
```

我们可以看到，从 XA 和 HZ 一共找到了 13 条可用路径。

## 2. 获得部分可用路径

在刚才的例子中，我们的可用路径其实并不是很多，只有 13 条。但是当我们网络中的链路稍多一些时，可用路径就会变得非常多。对于一个全互联的网络（即每两个节点之间都有直连的链路），任意两点之间的可用路径的数量值将为  $(n-2)!$  条。其值为节点数 -2 后的阶乘。如果节点数是 20 个，那么其任意两点之间的可用路径数量为 18!（18 的阶乘，约 6402 3737 0572 8000），这是一个非常庞大的数字。遍历这么多的可用路径，再快的计算机也许都无法在很短的时间内完成。

因此，我们希望获得的可用路径只是那些比最短路径稍微长一点的路径，这样我们就可以获得那些次优路径。我们可以修改上面例子中的代码为

```
>>> for path in nx.all_simple_paths(G, "XA", "HZ", 3):
...     print(path)
...
['XA', 'BJ', 'GZ', 'HZ']
['XA', 'BJ', 'SH', 'HZ']
['XA', 'GZ', 'HZ']
['XA', 'GZ', 'SH', 'HZ']
```

这里代码的含义是，我们希望获得从 XA 到 HZ 且经过的节点数量最多为 3（不包含起始节点）的所有路径。这次我们获得的可用路径只有 4 条。

## 3. 基于链路 cost 计算的可用路径

我们可以看到上面是基于网络节点的跳数来计算可用路径的，并不是基于链路的 cost 值来获取。大部分时候基于网络节点的跳数来计算可用路径是可以满足需要的，但我们还是希望能基于链路的 cost 值来获取更为有效、更为经济的可用路径。networkx 也提供了相应的算法。我们先看一下如下代码：

```
>>> def get_path_weight(G, path):
...     _weight = 0
...     for edge in nx.utils.pairwise(path):
...         _weight += G.edges[edge[0], edge[1]]["weight"]
...     return _weight
...
>>> for path in nx.shortest_simple_paths(G, "XA", "HZ", weight="weight"):
...     print(path, get_path_weight(G, path))
...
['XA', 'BJ', 'SH', 'HZ'] 3480
['XA', 'BJ', 'NJ', 'SH', 'HZ'] 3480
['XA', 'GZ', 'HZ'] 3600
['XA', 'BJ', 'WH', 'SH', 'HZ'] 3930
['XA', 'GZ', 'SH', 'HZ'] 4180
```



```

['XA', 'BJ', 'GZ', 'HZ'] 5500
['XA', 'BJ', 'GZ', 'SH', 'HZ'] 6080
['XA', 'GZ', 'BJ', 'SH', 'HZ'] 6580
['XA', 'GZ', 'BJ', 'NJ', 'SH', 'HZ'] 6580
['XA', 'GZ', 'BJ', 'WH', 'SH', 'HZ'] 7030

```

我们在上面的代码中先定义了一个函数，这个函数用来计算一条 path 路径的总 cost (networkx 中通常用 weight 参数表示) 值。

方法 `shortest_simple_paths` 会从最小 cost 的路径开始依次给出可用路径。由于这个方法返回的是一个生成器，因此我们可以根据需求随时停止可用路径的查找。

假设我们现在想获得 4 条可用路径，那么我们的代码可以这么来实现：

```

>>> from itertools import count
>>> counter = count(1)
>>> for c, path in zip(counter, nx.shortest_simple_paths(G, "XA", "HZ", weight=
    "weight")):
...     if c > 4:
...         break
...     print(path, get_path_weight(G, path))
...
['XA', 'BJ', 'SH', 'HZ'] 3480
['XA', 'BJ', 'NJ', 'SH', 'HZ'] 3480
['XA', 'GZ', 'HZ'] 3600
['XA', 'BJ', 'WH', 'SH', 'HZ'] 3930

```

获取这些可用路径对我们后续进行流量的优化和分析意义很大。关于如何使用 Python 来对网络进行优化的问题，我们将在第 14 章进行进一步的描述。

网络拓扑的路径计算是非常重要的内容，在网络优化、网络规划以及网络运维中都具有很有意义的应用。更多关于网络拓扑的算法可以参考 networkx 的文档 <https://networkx.github.io/documentation/stable/index.html>。

## 12.5 小结

本章主要介绍了两种在网络编程中的数据类型，它们是网络地址（包括 IPv4、IPv6 以及 MAC 地址）和网络拓扑。处理好这两种数据类型对网络编程是非常有用的。希望通过本章的例子，读者能掌握处理这种数据类型的基本处理方法。在第 13 章和第 14 章中，我们还会介绍它们的一些应用。我们可以在后续章节中加强一些印象。

随着本章的结束，我们的提高篇也随之结束了。在这一篇中，我们主要介绍了 Bash 和 Python 的编程入门。在 Python 的编程中，我们还用了几章的篇幅给大家介绍了一些常用的 Python 模块。笔者相信利用好这些模块是可以快速实现对网络设备的基本操作的，特别是对网络设备的配置层面的操作。在下一篇中，我们将用两章的内容来给大家提供几个案例，本书提供的案例也是广大网络工程师在日常工作中会遇到的情形。



## 第五篇 *Part 3*

# 案 例 篇

在前面的几章中，我们介绍了 Linux 下的一些文字处理工具，也介绍了 Python 语言和一些常用的 Python 模块。通过这些工具和 Python 模块，我们可以完成一些实用的小工具。

本篇希望通过两个案例的介绍让读者更加深入地了解 Python 相关的内容。这两个案例分别如下：

第一个案例，网络设备的配置管理。在网络运维阶段，大部分工作是管理和维护网络设备的配置。这个案例会涉及如何登录设备，以及如何进行配置的版本管理。

第二个案例，网络拓扑的处理与应用。除了网络管理和运维之外，网络设计是一项很重要的工作，网络设计有很多的内容，这里主要介绍网络拓扑的分析以及流量工程的仿真这两个问题。





## 网络设备的配置管理

我们在第 2 章中提到过，网络的核心任务是完成流量的转发。为了使网络能达到预期的流量、流向要求，我们可以从三个方面来实现这个目的（详见 2.4.1 节的内容）。其中第一种方法是通过修改网络设备的配置来实现目标，这种方式也是我们最常用的一种方法。基于这种方法的网络管理就会涉及大量的网络设备配置的管理。为了能很好地管理网络设备的配置，我们可以从以下两点入手。

首先，实现对网络设备配置的获取。获取当前网络设备的配置是 NetDevOps 的基础。我们可以通过配置获取的工作来完成程序和所有设备之间的基础通信。有了这个基础通道，我们就可以在后续获的更多的内容或者下发其他的配置。

其次，实现网络设备配置的版本管理。版本管理是非常重要的内容，通过配置文件的版本管理，我们就可以清楚地知道网络中发生了什么变化。在每次网络变更前后，离线保存设备的配置是非常有必要的。

### 13.1 环境的准备

#### 13.1.1 测试拓扑说明

在开始这个案例之前，笔者设计了一个典型的小型数据中心的网络拓扑（见图 13-1）。在这个拓扑中，每台设备都会连接到带外网中，在带外网中有一台 Linux 服务器，我们所有的代码都运行在这台 Linux 服务器上。在这个网络中存在三个厂家的设备，这些设备都是通过软件平台来实现的。它们包括 Juniper vMX、Cisco NX-OSv 以及 Arista vEOS。在这个网络中，出口路由、核心交换机以及接入交换之间使用了 OSPF 协议，出口路由器和外



部的 SP 设备之间使用了 BGP 协议。不过这个拓扑中省略了防火墙和 NAT 设备。

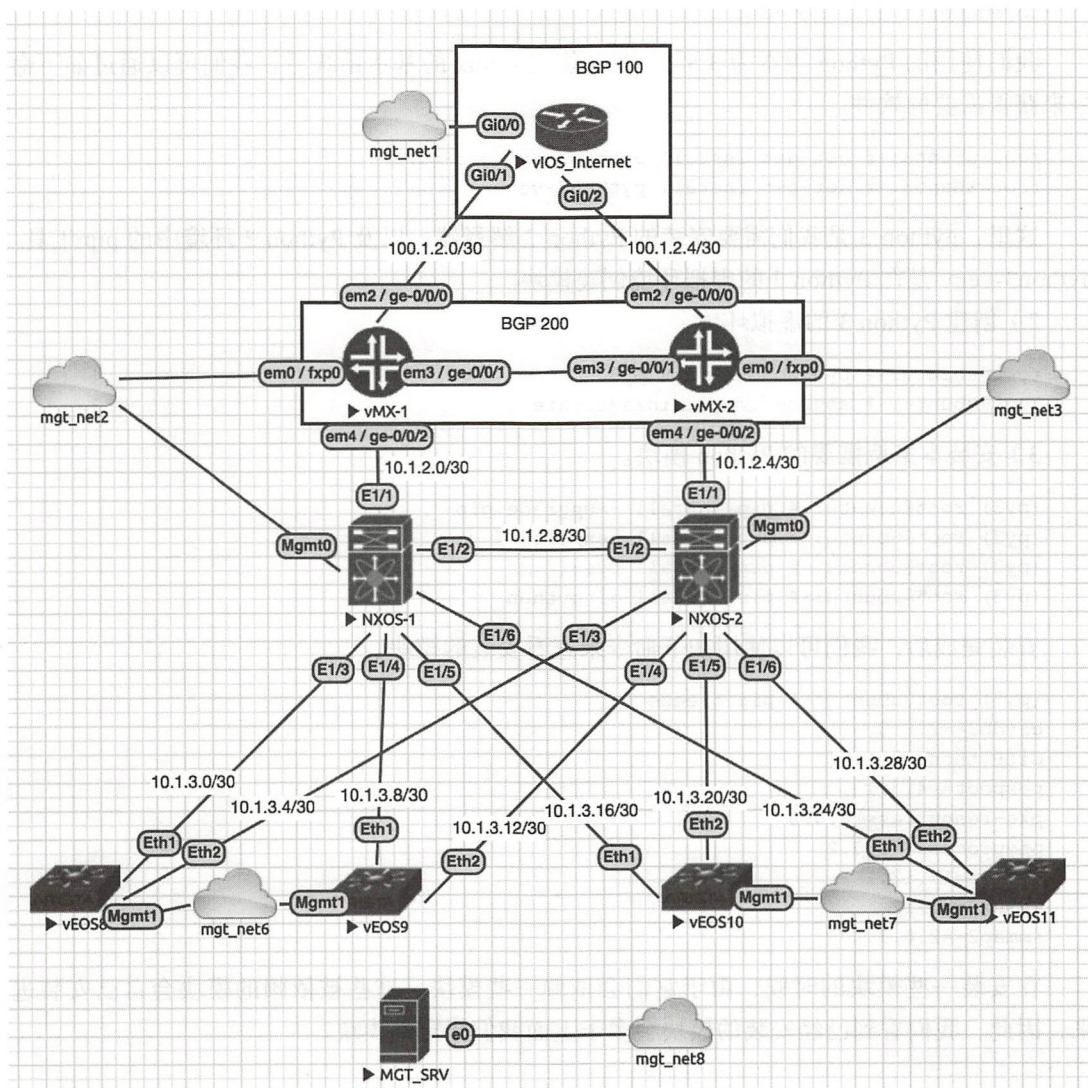


图 13-1 拓扑图

### 13.1.2 Linux 服务器的准备

#### 1) 安装相应的软件包。

在这个测试环境中，我们使用的是 Ubuntu 16.04 Server 版本的 Linux 发行版。

```
root@ubuntu:~# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
```

```

Description:    Ubuntu 16.04.2 LTS
Release: 16.04
Codename:       xenial

```

我们使用的 Python 版本为 3.5.2。在安装完 Linux 的基本系统后，我们可以通过如下命令来获得需要的软件。

```

root@ubuntu:~# apt-get install python3 python3-pip
root@ubuntu:~# apt-get install python3-venv

```

这里，python 3 是我们需要安装的 Python 3 解释器，以及 Python 3 环境中的 pip 工具。python3-venv 是在 Python 3 的虚拟环境创建模块。

2) 创建 Python 3 的虚拟环境。

```

root@ubuntu:~# python3 -m venv py3
root@ubuntu:~# source ./py3/bin/activate

```

3) 安装本案例需要的 Python 模块。

```

(py3) root@ubuntu:~# pip install --upgrade pip
(py3) root@ubuntu:~# pip install pexpect
(py3) root@ubuntu:~# pip install pyaml
(py3) root@ubuntu:~# pip install gitpython

```

安装完成后，我们可以使用如下命令来查看安装的模块。

```

(py3) root@ubuntu:~# pip freeze
gitdb2==2.0.3
GitPython==2.1.8
pexpect==4.3.1
pkg-resources==0.0.0
ptyprocess==0.5.2
pyaml==17.12.1
PyYAML==3.12
smmap2==2.0.3

```

在安装一些模块的时候，由于它们会依赖一些模块，这些被依赖的模块会自己安装进来，因此，我们看到已经安装的模块会多于上面我们指定的模块。

## 13.2 网络设备的配置获取

在这个案例中，每台设备都支持 SSH 登录，部分的设备还支持 Telnet 登录，每一台设备都支持 Telnet 的方式登录到设备的 Console 接口（由虚拟化平台提供的功能）。虽然在实验环境中设备都支持 NETCONF 议，但是这里我们还是以使用 CLI 方式为主。

### 13.2.1 登录网络设备

在登录设备之前，我们使用 YAML 格式的文件保存所有设备通过带外管理登录的信



息。YAML 文件（YAML 文件相关内容请参考 9.3 节）的内容如下：

```
devices:
  - vIOS:
    hostname: vIOS_internet
    mgt_ip: 172.20.100.10
    vendor: Cisco
    OS_type: IOS
    username: admin
    password: lab123
  - vMX-1:
    hostname: vMX-1
    mgt_ip: 172.20.100.11
    vendor: Juniper
    OS_type: JUNOS
    username: admin
    password: lab123
  - vMX-2:
    hostname: vMX-2
    mgt_ip: 172.20.100.12
    vendor: Juniper
    OS_type: JUNOS
    username: admin
    password: lab123
  - NXOS-1:
    hostname: NXOS-1
    mgt_ip: 172.20.100.13
    vendor: Cisco
    OS_type: NXOS
    username: admin
    password: Admin@123
  - NXOS-2:
    hostname: NXOS-2
    mgt_ip: 172.20.100.14
    vendor: Cisco
    OS_type: NXOS
    username: admin
    password: Admin@123
  - vEOS8:
    hostname: vEOS8
    mgt_ip: 172.20.100.20
    vendor: Arista
    OS_type: EOS
    username: admin
    password: lab123
  - vEOS9:
    hostname: vEOS9
    mgt_ip: 172.20.100.21
    vendor: Arista
    OS_type: EOS
    username: admin
```

```

password: lab123
- vEOS10:
  hostname: vEOS10
  mgt_ip: 172.20.100.22
  vendor: Arista
  OS_type: EOS
  username: admin
  password: lab123
- vEOS11:
  hostname: vEOS11
  mgt_ip: 172.20.100.23
  vendor: Arista
  OS_type: EOS
  username: admin
  password: lab123

```



**注意** 通过一个文件来保存所有设备的用户名和密码是非常不安全的，在这个例子中，我们简化了认证信息的内容。

在这个例子中，我们使用 `pexpect` 模块来完成和设备的交互操作。首先，我们实现一个基本的登录设备的代码。

```

#!/usr/bin/env python
#coding: utf-8
import pexpect
from pexpect import EOF, TIMEOUT

# 使用 Linux 的 ssh 命令来连接设备
# StrictHostKeyChecking=no 和 UserKnownHostsFile=/dev/null 是为了不保存也不检查网络
# 设备上的指纹信息
# 这个函数返回了一个 pexpect 的子进程
def ssh_connect(username, address, port):
    ssh_command = 'ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null'
    -l %s %s -p %d' % (
        username, address, port)
    return pexpect.spawn(ssh_command)

# 定义一个设备的基本对象
class Device(object):

    def __init__(self, device):
        # 这个类初始化时使用的变量是一个字典类型的变量。这个字典包含以下几个主键：
        # hostname, mgt_ip, username, password, port
        self.hostname = device.get("hostname")
        self.mgt_ip = device.get("mgt_ip")
        self.username = device.get("username")
        self.password = device.get("password")
        self.port = device.get("port", 22)
        self.expect_list = []

```

```

def connect(self, timeout=30):
    # 定义一个方法，这个方法用于创建一个 SSH 连接
    self.c = ssh_connect(self.username, self.mgt_ip, self.port)
    self.c.delaybeforesend = 0.10
    return self.c

def login(self, prompt=r">|#|$]\s?$"):
    # 定义登录设备的基本方法
    # 在登录设备时，遇到的设备给的提示信息通常为 username: 或者 login:, 输入用户名
    # 提示信息为 password: 时，需要输入密码
    # 在登录完成设备后，需要获取设备的提示符。网络设备的提示符通常为
    # ">"、"#" 以及 "$"
    self.expect_list = []
    self.expect_list.append(r"(?i)username[:]?\s*$")
    self.expect_list.append(r"(?i)login[:]?\s*$")
    self.expect_list.append(r"(?i)password[:]?\s*$")
    self.expect_list.append(prompt)
    for _ in range(0, 2):
        result = []
        try:
            i = self.c.expect(self.expect_list, timeout=5)
            result.append(i)
            result.append(str(self.c.before))
            result.append(str(self.c.after))
            if i < 2:
                self.c.sendline(self.username)
            elif i == 2:
                self.c.sendline(self.password)
            elif i == 3:
                break
        except EOF:
            break
        except TIMEOUT:
            print("connect to %s timeout" % self.hostname)
            break
    # 当设备一直停留在用户名或者是密码的提示符时，那么很可能是用户名和密码出现了问题
    # 在这里使用 log 的方式显示在标准输出上或记录在一个 log 文件中更好。这里为了简化就直接
    # 使用 print 直接输出
    if result[0] < 3:
        print("username or password error")
        return result

    # 大部分网络设备对输出的结果都会提供默认的分屏功能。在代码执行时，
    # 我们并不需要这个功能。我们需要设置终端的长度为 0
    self._set_terminal_length_zero()
    return result

# 这里定义的函数，并没有完成其具体实现的代码。具体的实现，可以在子类中完成
def get_config(self):
    pass

```



```

# 定义一个退出的方法。当退出的时候或者这个对象被删除的时候，关闭 SSH 的子进程
def logout(self):
    if self.c:
        self.c.terminate()

def __del__(self):
    self.logout()

# 这个子类是专门针对 JUNOS 的子类。在这个类中，需要对父类 Device 进行部分修改。使其符合 JUNOS
class JUNOS(Device):

    def __init__(self, device):
        # 先执行其父类 Device 的初始化方法
        super(JUNOS, self).__init__(device)
        # 由于 JUNOS 的设备提示符是确定的，其格式为“登录的用户名@主机名”，最后使用“>”或
        # “#”结尾。其中“>”表示操作模式，而“#”表示配置模式
        self.prompt = self.username + "@" + self.hostname + "[>|#]"

    def login(self, prompt=""):
        # 对登录设备的方法也做了简单的修改。只是修改了默认的提示符为 JUNOS 的提示符
        if not prompt:
            prompt = self.prompt
        return super(JUNOS, self).login(prompt)

    def get_config(self):
        # 这个方法在父类中并没有实现，在 JUNOS 这个子类中来具体实现
        # 首先定义设备的提示符的信息
        self.expect_list = []
        self.expect_list.append(self.prompt)
        result = []
        # 通过 show config 命令来获取设备的配置信息
        self.c.sendline("show config | no-more")
        try:
            i = self.c.expect(self.expect_list, timeout=5)
            if i == 0:
                result.append(i)
                result.append((self.c.before + self.c.after).decode())
        except EOF:
            pass
        except TIMEOUT:
            print("session timeout")
        return result

# 当这段代码被直接执行的时候，这里是程序的入口
if __name__ == "__main__":
    # 定义一个测试用的设备信息
    d = {"hostname": "vMX-1",
         "mgt_ip": "172.20.100.11",
         "username": "admin",
         "password": "lab123"}
    # 初始化 JUNOS 这个类

```

```

conn = JUNOS(d)
# 连接到设备上
conn.connect()
# 通过用户名和密码登录设备
conn.login()
# 获取设备的配置信息
print(conn.get_config())

```

在之前的内容中，我们很少在编写的代码中使用面向对象的编程方法（虽然在 Python 中一切皆对象），但本章将会使用面向对象的编程方法。因此，在本章看到的代码感觉会比之前的要复杂一些。

这个例子有两个类，一个是 Device 类，另一个是 JUNOS 这个类，而 JUNOS 继承了 Device 类的子类。大部分功能在 Device 这个基类中已经实现了。在 JUNOS 类中修改了基类中的一些功能。

### 13.2.2 处理多厂家问题

大部分情况下，我们的网络中存在多个不同的厂家共存的情况。不同的网络设备之间是存在一些小的差异的，因此，我们需要对不同的厂家定义不同的类。现在我们添加一个 NXOS 的类，用于处理 Cisco NXOS 设备。这个类的代码如下：

```

class NXOS(Device):

    def __init__(self, device):
        super(NXOS, self).__init__(device)
        self.prompt = self.hostname + ">|#|\s?"

    def login(self, prompt=""):
        if not prompt:
            prompt = self.prompt
        super(NXOS, self).login(prompt)
        self._set_terminal_length_zero()

    def _set_terminal_length_zero(self):
        self.c.sendline("terminal length 0")
        try:
            i = self.c.expect(self.prompt)
        except EOF:
            pass
        except TIMEOUT:
            print("session timeout")

    def get_config(self):
        self.expect_list = []
        self.expect_list.append(self.prompt)
        result = []
        self.c.sendline("show running-config")
        try:

```

```

        i = self.c.expect(self.expect_list, timeout=5)
        if i == 0:
            result.append(i)
            result.append((self.c.before + self.c.after).decode())
    except EOF:
        pass
    except TIMEOUT:
        print("session timeout")
    return result

```

现在我们已经为 NXOS 单独写了一个类，而这个类可以处理 NXOS 设备。我们在代码中实现了登录的方法 (login)，也实现了设置终端长度的内容，还实现了获取设备配置的代码。

后续再增加新的设备的时候，我们也可以使用类似方法来添加新的设备代码。但是，我们如何来组织我们的代码结构呢？和第一段代码一样，把所有内容都放在一个文件中，从最终的实现效果而言，这种方法是可行的，但是并不适合后续的代码维护。

我们可以这样来组织我们的代码。首先，我们将所有和设备交互的代码放在一个文件夹中，这个文件夹名为 NetDevices。然后，在这个目录下根据不同的类创建不同的文件名。命令如下：

```

# mkdir NetDevices
# cd NetDevices
# touch __init__.py
# touch Device.py
# touch JUNOS.py
# touch NXOS.py

```

接下来，我们把之前的代码分别放在这几个文件中。其中函数 ssh\_connect 放在 Device.py 文件中。

这里我们列出这个目录的目录结构：

```

# tree NetDevices/
NetDevices/
|-- Device.py
|-- __init__.py
|-- JUNOS.py
`-- NXOS.py

0 directories, 4 files

```

当一个目录需要成为一个 Python 模块时，需要在这个目录下创建文件 \_\_init\_\_.py，这个文件会在导入这个模块时执行。在这个例子中，\_\_init\_\_.py 现在的代码如下：

```

from NetDevices.JUNOS import JUNOS
from NetDevices.NXOS import NXOS

```

这样的代码可以减少模块中的命名空间的长度。

在 JUNOS.py 和 NXOS.py 这两个文件中，在一开始需要增加如下这行代码（因为我们



在代码中需要继承 Device 这个类，因此我们需要导入 Device 类的代码。完整代码见 13.2.3 节中的例子)。

```
from NetDevices.Device import Device
```

现在，我们已经把不同的设备类型分成了不同的文件来编写代码。使用这个自己写的模块时的代码如下：

```
#!/usr/bin/env python
#coding:utf-8
from NetDevices import JUNOS
d = {"hostname": "vMX-1",
     "mgt_ip": "172.20.100.11",
     "username": "admin",
     "password": "lab123"}

conn = JUNOS(d)
conn.connect()
conn.login()
print(conn.get_config())
```

这样的代码看上去还不错。但是，也许读者还记得我们在本章的一开始时记录了这次实验环境中的一些设备基本信息，这些基本信息是包含了设备的厂家信息和操作系统信息的。我们现在来实现通过这些基本信息，让代码自己来选择使用哪一个具体的类来完成设备操作。

我们需要修改文件 `__init__.py` 中的代码，具体如下：

```
#coding:utf-8
def DeviceHandler(device):
    # 从字典中获取设备的 OS 类型
    os_type = device.get("OS_type", "JUNOS")

    # 尝试去加载模块中的子模块。这个模块的命名规则为 NetDevices."OS_type"。这里的 OS_type
    # 为目录 NetDevices 下的文件名，比如 JUNOS.py 或者 NXOS.py 这样的文件名
    try:
        device_module = __import__("NetDevices.%s" % os_type)
    except ImportError:
        pass

    # 使得 device_module 值为 "NetDevices.os_type"
    device_module = getattr(device_module, os_type)

    # 获得类的变量。这里类的名称和 os_type 的值相同
    device_class = getattr(device_module, os_type)
    # 最后返回类的实例
    return device_class(device)
```

通过这样的修改，后续我们在使用这个模块的时候可以在字典中携带网络设备操作系

统的值，这样就可以让代码帮你找到正确的模块和类了。另外，以后再添加新的网络操作系统，我们也不用修改这个 `__init__.py` 文件，只需要在目录中添加适当的文件和代码就可以了。

现在，我们就可以使用这样的代码来完成配置的获取了：

```
#!/usr/bin/env python
#coding: utf-8

from NetDevices import DeviceHandler

d = {"hostname": "vMX-1",
     "mgt_ip": "172.20.100.11",
     "username": "admin",
     "password": "lab123",
     "OS_type": "JUNOS"}

conn = DeviceHandler(d)
conn.connect()
r = conn.login()
print(conn.get_config())
```

### 13.2.3 处理并行问题

对于程序而言，在和网络设备进行通信时，大部分的时间消耗在等待网络设备给出响应，这也是大部分的基于网络的程序面临的一个情况。在本书中，之前的所有程序都是按照一定的顺序执行的。如果按照这种方式，当我们从大量的网络设备上获取很多信息时，程序需要执行很长一段时间，并且它是一台设备一台设备按顺序执行的。举个例子：现在我们要获取网络中 100 台设备的配置信息。我们通常的做法如下：先登录设备，然后输入“`show running-config`”（不同厂家会有不同的命令），最后保存这些配置信息。如果获取每台设备的配置信息需要 2~3s，那么获取 100 台设备的配置信息就需要 200~300s。如果是 1000 台甚至是 10 000 台设备呢，恐怕消耗的时间就更多了。其实，在程序执行的过程中，大量的时间消耗在等待设备给出数据，程序只是在那里等待，并没有做任何事情。为了解决这个问题，我们基本上有两种解决思路。一种思路是，我们为每一台设备都启用一个进程，这个进程被分配单独一个 CPU 来完成它。但是，这种操作太消耗物理资源，会让那些 CPU 长期处于一个闲置等待的状态。另一种思路是，我们可以在网络设备返回数据的这个时间里，让程序去处理其他网络设备的交互。其实，我们使用的大部程序都是用这种方法来处理问题的。

Python 3.4 以后版本加入了协程和异步 I/O（`asyncio`）的概念。对于 `pexpect` 模块，在 `pexpect 4.0` 版本之后，且 Python 3.4 以上就支持了异步 I/O。下面我们来修改之前的代码，让其支持异步 I/O 模式。

我们需要对所有需要进行异步操作的函数进行修改。正如前面讨论的，和网络设备进

行交互时，向设备发送完命令后就是长时间地等待设备的响应（对于 CPU 执行代码而言，等待几十毫秒甚至几毫秒都是非常漫长的）。在 `pexpect` 模块中，`expect` 方法就是等待设备给响应的方法。因此，我们需要对所有使用 `expect` 方法的地方进行修改。

在文件 `Device.py` 中，我们需要修改的是下面代码中加粗与斜体的部分。

```
#!/usr/bin/env python
#coding: utf-8
import pexpect
from pexpect import EOF, TIMEOUT

def ssh_connect(username, address, port):
    ssh_command = 'ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null \
        -l %s %s -p %d' % (
        username, address, port)
    return pexpect.spawn(ssh_command)

class Device(object):

    def __init__(self, device):

        self.hostname = device.get("hostname")
        self.mgt_ip = device.get("mgt_ip")
        self.username = device.get("username")
        self.password = device.get("password")
        self.port = device.get("port", 22)
        self.expect_list = []

    def connect(self, timeout=30):
        self.c = ssh_connect(self.username, self.mgt_ip, self.port)
        self.c.delaybeforesend = 0.10
        return self.c

    async def login(self, prompt=r">|#|$]\s?$"):
        self.expect_list = []
        self.expect_list.append(r"(?i)username[:]?\s*$")
        self.expect_list.append(r"(?i)login[:]?\s*$")
        self.expect_list.append(r"(?i)password[:]?\s*$")
        self.expect_list.append(prompt)
        for _ in range(0, 2):
            result = []
            try:
                # 修改前的内容
                # i = self.c.expect(self.expect_list, timeout=5)
                i = await self.c.expect(self.expect_list, timeout=5, async_=True)
                result.append(i)
                result.append(str(self.c.before))
                result.append(str(self.c.after))
                if i < 2:
                    self.c.sendline(self.username)
```



```

        elif i == 2:
            self.c.sendline(self.password)
        elif i == 3:
            break
    except EOF:
        break
    except TIMEOUT:
        print("connect to %s timeout" %self.hostname)
        break

    if result[0] < 3:
        print("username or password error")
        return result

    return result

def get_config(self):
    pass

def logout(self):
    if self.c:
        self.c.terminate()

def __del__(self):
    self.logout()

```

JUNOS.py 与 NXOS.py 这两个文件中也有需要使用异步 I/O 的方法需要修改。

下面是 JUNOS.py 文件的内容。

```

#coding: utf-8
from NetDevices.Device import Device
from pexpect import EOF, TIMEOUT

class JUNOS(Device):

    def __init__(self, device):
        super(JUNOS, self).__init__(device)
        self.prompt = self.username + "@" + self.hostname + "[>|#]"

    async def login(self, prompt=""):
        if not prompt:
            prompt = self.prompt
        await super(JUNOS, self).login(prompt)

    async def get_config(self):
        self.expect_list = []
        self.expect_list.append(self.prompt)
        result = []
        self.c.sendline("show config | no-more")
        try:
            # 修改前的内容

```

```

# i = self.c.expect(self.expect_list, timeout=5)
i = await self.c.expect(self.expect_list, timeout=5, async_=True)
if i == 0:
    result.append(i)
    result.append((self.c.before + self.c.after).decode())
except EOF:
    pass
except TIMEOUT:
    print("session timeout")
return result

```

下面是 NXOS.py 文件的内容。

```

# coding:utf-8
from NetDevices.Device import Device
from pexpect import EOF, TIMEOUT

class NXOS(Device):

    def __init__(self, device):
        super(NXOS, self).__init__(device)
        self.prompt = self.hostname + ">|#|\s?"

    async def login(self, prompt=""):
        if not prompt:
            prompt = self.prompt
        await super(NXOS, self).login(prompt)
        await self._set_terminal_length_zero()

    async def _set_terminal_length_zero(self):
        self.c.sendline("terminal length 0")
        try:
            i = await self.c.expect(self.prompt, async_=True)
        except EOF:
            pass
        except TIMEOUT:
            print("session timeout")

    async def get_config(self):
        self.expect_list = []
        self.expect_list.append(self.prompt)
        result = []
        self.c.sendline("show running-config")
        try:
            i = await self.c.expect(self.expect_list, timeout=5, async_=True)
            if i == 0:
                result.append(i)
                result.append((self.c.before + self.c.after).decode())
        except EOF:
            pass
        except TIMEOUT:

```

```

        print("session timeout")
    return result

```

对于上述修改的代码，我们可以这样快速地理解。首先，找到那些需要 I/O 等待的代码位置，在 `pexpect` 中就是 `expect` 方法。我们先修改 `expect` 的默认参数，使得其支持协程的处理方式。然后在这个方法前加入关键字“`await`”，使得当程序运行到这里时可以被协程所调度。协程的事件管理器会把这个关键字后的语句先搁置起来，而去运行其他方法（其他异步函数）中的代码。等到再次回到这个位置时，将继续往后执行，直到再次遇到“`await`”关键字修饰的语句。这样，我们就可以不浪费 CPU 的时间让 CPU 尽量多地做一些事情。

最后，在调用这些异步方法（异步函数）时，也存在一些差异。具体的代码如下：

```

#!/usr/bin/env python
#coding: utf-8
import asyncio

from NetDevices import DeviceHandler
# 定义两台设备的基本信息, d1 和 d2
d1 = {"hostname": "vMX-1",
      "mgt_ip": "172.20.100.11",
      "username": "admin",
      "password": "lab123",
      "OS_type": "JUNOS"}

d2 = {"hostname": "vMX-2",
      "mgt_ip": "172.20.100.12",
      "username": "admin",
      "password": "lab123",
      "OS_type": "JUNOS"}

# 定义一个支持异步的函数
async def get_config(device):
    conn = DeviceHandler(device)
    conn.connect()
    # 登录时需要等待设备返回
    await conn.login()
    # 获取设备的配置信息
    r = await conn.get_config()
    print(r)

# 创建一个异步事件循环器
loop = asyncio.get_event_loop()
# 执行两台设备获取配置的任务
loop.run_until_complete(asyncio.gather(get_config(d1), get_config(d2)))

```

现在我们通过 `yaml` 文件中记录的设备信息来获取设备的配置文件。

```

#!/usr/bin/env python
#coding: utf-8

```



```

import asyncio
import yaml
import sys
from NetDevices import DeviceHandler

async def get_config(device):
    conn = DeviceHandler(device)
    conn.connect()
    await conn.login()
    r = await conn.get_config()
    for line in r[1].split("\r\n"):
        print(line)

try:
    yaml_cfg = sys.argv[1]
except IndexError:
    print("please give yaml configure file")
    sys.exit(1)

f = open(yaml_cfg)
deviceinfos = yaml.load(f.read())

print(deviceinfos)

loop = asyncio.get_event_loop()
tasks = []

# 创建任务列表
for device in deviceinfos.get("devices"):
    tasks.append(loop.create_task(get_config(device)))
# 执行任务列表
loop.run_until_complete(asyncio.wait(tasks))

```

### 13.3 网络设备的配置版本管理

在前面的部分，我们已经完成了获取设备配置的功能。我们需要将这些配置保存在磁盘中。对于设备的配置文件，我们需要有良好的机制来管理其版本信息。对于版本的管理，我们最少要满足这几点要求：

- ❑ 设备的配置文件要有明确的保存时间；
- ❑ 设备的配置文件需要保留其历史版本的全内容；
- ❑ 通过设备的信息能快速地找到相应的配置文件；
- ❑ 能记录每个版本的修改信息。

这里我们使用 git 工具来管理设备配置文件。git 是一个分布式的版本管理工具，其功能非常丰富且强大。现在大量的程序开发中的版本管理都使用 git 作为其版本管理的工具。

### 13.3.1 用 git 创建一个本地设备配置管理仓库

首先，我们创建一个目录。这个目录用于设备的配置信息。

```
# mkdir device_cfg
```

使用 git 命令初始化这个文件夹为一个 git 仓库。

```
# git init device_cfg/
```

这样我们就完成了一个本地 git 仓库的创建工作，并且在目录 device\_cfg/ 下会创建一个 .git 的子目录。

### 13.3.2 保存设备配置文件到本地仓库

现在，我们需要通过 13.2.3 节中的代码来获取设备的配置文件，然后保存到本地的 git 仓库中。

```
#!/usr/bin/env python
#coding: utf-8
import asyncio
import yaml
import sys
from NetDevices import DeviceHandler
from git import Repo
import time
```

```
FILEPATH = "/root/device_cfg/"
async def get_config(device):
    hostname = device.get("hostname")
    conn = DeviceHandler(device)
    conn.connect()
    await conn.login()
    r = await conn.get_config()
```

```
    # 保存文件到 git 仓库的目录中
    file_name = FILEPATH + hostname
    open(file_name, "w").write(r[1])
    print("%s is saved" %file_name)
```

```
deviceinfos = {}
try:
    yaml_cfg = sys.argv[1]
```

```
except IndexError:
    print("please give yaml configure file")
    sys.exit(1)
```

```
f = open(yaml_cfg)
deviceinfos = yaml.load(f.read())
```

```

loop = asyncio.get_event_loop()
tasks = []
for device in deviceinfos.get("devices"):
    tasks.append(loop.create_task(get_config(device)))

loop.run_until_complete(asyncio.wait(tasks))

# 向 git 仓库提交保存的文件
localtime = time.asctime(time.localtime(time.time()))
repo = Repo(FILEPATH)
repo.index.add("")
repo.index.commit("Configs auto saving at %s" %localtime)

```

在这段代码中，加粗和斜体的部分是和 13.2.3 节中最后一段代码有区别的内容。这段修改后的代码实现的功能是将从网络设备上获取的配置文件保存到 git 本地仓库中。

git 会自行检查哪些文件被修改过，并只提交那些已经被修改过的文件。

### 13.3.3 使用 git 检查版本信息

对于保存在 git 仓库中的内容，我们可以通过 git 命令来查看。所有这些命令都需要在 git 仓库所在的目录下执行。这里列举一些常用的命令。

#### (1) 列出当前仓库内的版本信息

```

# git log --oneline
0b97eaf Configs auto saving at Wed Dec 27 20:38:07 2017
bbbbde5 devices configure were saved in 2017-12-27
60a5242 device configure was saved in 2017-12-27

```

#### (2) 查看某个文件的版本信息

```

# git log --oneline vEOS8
60a5242 device configure was saved in 2017-12-27
# git log --oneline NXOS-1
0b97eaf Configs auto saving at Wed Dec 27 20:38:07 2017
bbbbde5 devices configure were saved in 2017-12-27
60a5242 device configure was saved in 2017-12-27

```

我们可以发现，如果文件没有发生变化，那么这个文件的版本信息将不会发生变化。

#### (3) 查看某个文件的内容和上一个版本之间的差异

我们修改了 vEOS8 设备的配置，然后进行配置的备份。下面先来查询一下，这次修改了哪些配置。

```

$ git log --oneline vEOS8
38d9845 Configs auto saving at Wed Dec 27 23:41:49 2017
60a5242 device configure was saved in 2017-12-27

```

使用 git show 命令来比较当前版本和上一个版本之间的差异。

```

# git show 38d9845 vEOS8

```



```
commit 38d984599920a3058e3bdc5bc92e65b328eb1310
Author: Your Name <you@example.com>
Date:   Wed Dec 27 23:41:49 2017 +0800
```

```
Configs auto saving at Wed Dec 27 23:41:49 2017
```

```
diff --git a/vEOS8 b/vEOS8
index f564166..14afd7f 100644
--- a/vEOS8
+++ b/vEOS8
@@ -25,8 +25,6 @@ interface Ethernet2
     ip address 10.1.3.6/30
!
+ interface Ethernet3
-   no switchport
-   ip address 100.1.2.97/30
!
+ interface Ethernet4
!
```

命令中的 38d9845 是版本的 Hash 值的短形式，其后面的参数是需要查看的文件名。上面加粗部分是当前版本和上一个版本之间的区别。这让我们快速地知道了某个版本和上个版本之间的区别。

#### (4) 查看文件中每一行来自哪个版本

现在我们在 vEOS9 这台设备上增加一些配置，然后备份一次设备的配置文件。

```
git blame vEOS9
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 1) show running-config
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 2) ! Command: show running-config
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 3) ! device: vEOS9 (vEOS, EOS-4.18.1F)
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 4) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 5) ! boot system flash:/vEOS-lab.swi
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 6) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 7) transceiver qsfp default-mode 4x10G
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 8) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 9) hostname vEOS9
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 10) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 11) spanning-tree mode mstp
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 12) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 13) aaa authorization exec default local
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 14) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 15) no aaa root
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 16) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 17) username admin privilege 15 role
network-admin secret sha512 $6$FBfSwhmBCHL8PcAb$kkzZNLl3jtgFci3woU9Zas0K8ZVWD5zgLIqZdC
NV/qI8irG30Y5adOsrhp91C.PA4C/KYvRlPZeWcYPckUMMy/
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 18) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 19) interface Ethernet1
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 20)     no switchport
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 21)     ip address 10.1.3.10/30
```

```

^60a5242 (Your Name 2017-12-27 19:53:12 +0800 22) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 23) interface Ethernet2
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 24)     no switchport
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 25)     ip address 10.1.3.14/30
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 26) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 27) interface Ethernet3
7b312819 (Your Name 2017-12-27 23:57:10 +0800 28)     no switchport
7b312819 (Your Name 2017-12-27 23:57:10 +0800 29)     ip address 100.1.2.105/30
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 30) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 31) interface Ethernet4
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 32) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 33) interface Ethernet5
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 34) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 35) interface Ethernet6
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 36) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 37) interface Ethernet7
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 38) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 39) interface Loopback0
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 40)     ip address 10.1.0.11/32
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 41) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 42) interface Management1
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 43)     ip address 172.20.100.21/16
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 44) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 45) ip routing
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 46) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 47) router ospf 1
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 48)     network 10.1.0.11/32 area 0.0.0.2
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 49)     network 10.1.3.8/30 area 0.0.0.2
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 50)     network 10.1.3.12/30 area 0.0.0.2
7b312819 (Your Name 2017-12-27 23:57:10 +0800 51)     network 100.1.2.104/30 area 0.0.0.2
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 52)     max-lsa 12000
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 53) !
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 54) end
^60a5242 (Your Name 2017-12-27 19:53:12 +0800 55) vEOS9#

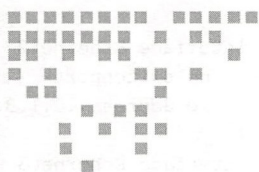
```

在这里的输出结果中，加粗部分是另一个版本中增加的内容。

git 工具还有很多命令可以用于管理文件的版本信息。除了命令行的工具之外，我们也可以使用 git 图形化工具管理和查看文件，限于篇幅，这里我们就不再一一举例。更多关于 git 工具的内容，我们可以参考 <https://git-scm.com/book/zh/v2>。

## 13.4 小结

在本章中，我们其实只完成了备份网络设备的配置文件这个小功能。我们利用了 Python 3 中的异步 I/O 实现了操作网络设备的并行问题，还使用了 git 工具对网络设备的配置文件进行了版本管理。这个例子其实并不复杂，相信读者能根据例子开发一个满足自己工作需要的配置备份工具。



## Chapter 14 第 14 章

# 网络拓扑的处理与应用

第 10 章主要介绍了设备配置的获取与管理。这些工作主要在网络运维阶段涉及较多。我们在 12.4 节中简单介绍过网络拓扑的处理，本章将通过一个案例来介绍如何处理网络的拓扑和如何在网络设计与规划中来分析拓扑。

## 14.1 环境的准备

### 14.1.1 测试拓扑说明

在开始这个案例之前，笔者设计了一个广域网的网络拓扑（见图 14-1）。在这个拓扑中，有一台设备连接到了一台 Linux 服务器上，我们所有的代码都运行在这台 Linux 服务器上。在这个网络中存在两个厂家的设备，这些设备都是通过软件平台实现的。它们包括 Juniper vMX 和 Cisco IOS XRv。在这个网络中，所有的网络设备运行 ISIS 协议作为 IGP 协议。全网所有设备都运行 BGP 协议，BGP 的 AS 号为 4000。

表 14-1 给出了所有设备的 Loopback 地址信息。表 14-2 给出了所有链路的地址信息。

### 14.1.2 Linux 服务器的准备

#### 1. 安装相应的软件包

在这个测试环境中，我们使用的是 Ubuntu 16.04 Server 版本的 Linux 发行版。

```
root@ubuntu:~# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
```



Description: Ubuntu 16.04.3 LTS  
Release: 16.04  
Codename: xenial

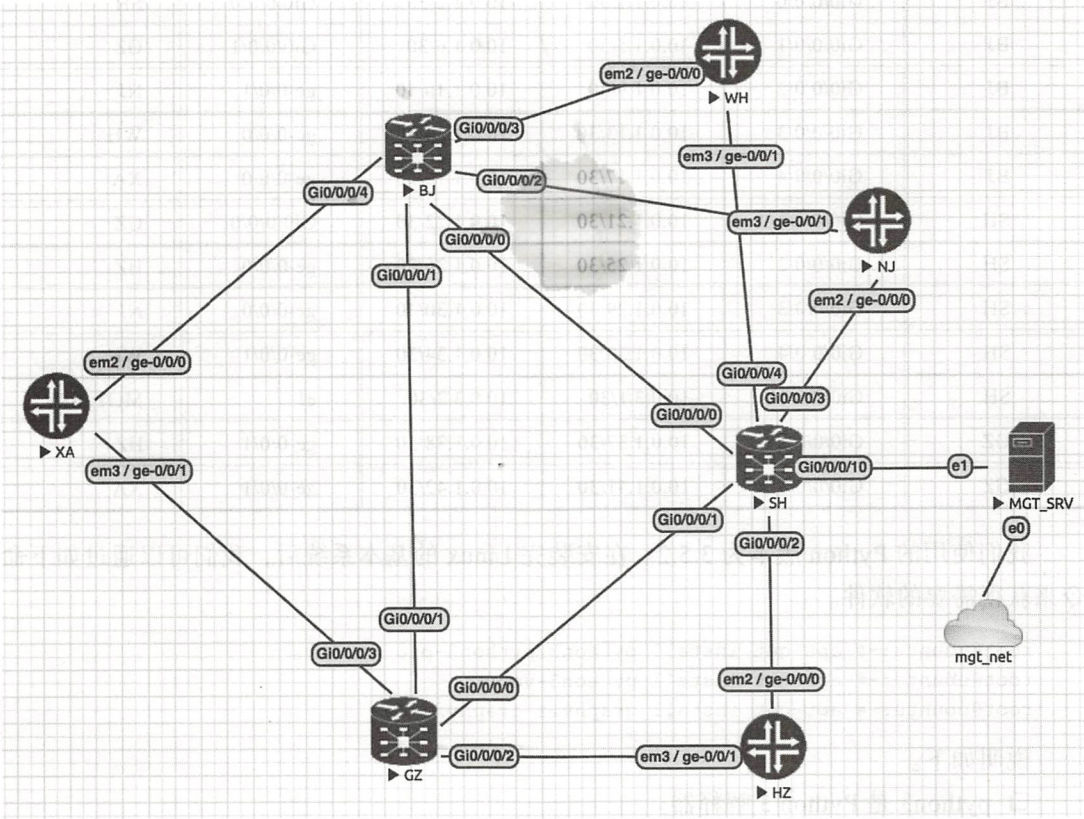


图 14-1 拓扑图

表 14-1 设备 Loopback 地址

设备名	Loopback0	ISIS NET 地址
BJ	10.0.0.1	49.0001.0100.0000.0001.00
SH	10.0.0.2	49.0001.0100.0000.0002.00
GZ	10.0.0.3	49.0001.0100.0000.0003.00
HZ	10.0.0.4	49.0001.0100.0000.0004.00
NJ	10.0.0.5	49.0001.0100.0000.0005.00
WH	10.0.0.6	49.0001.0100.0000.0006.00
XA	10.0.0.7	49.0001.0100.0000.0007.00

表 14-2 链路 IP 地址分配

A 端设备名	A 端接口	A 端地址	Z 端地址	Z 端接口	Z 端设备名
BJ	Gi0/0/0/0	10.0.1.1/30	10.0.1.2/30	Gi0/0/0/0	SH
BJ	Gi0/0/0/1	10.0.1.5/30	10.0.1.6/30	Gi0/0/0/1	GZ
BJ	Gi0/0/0/2	10.0.1.9/30	10.0.1.10/30	ge-0/0/1	NJ
BJ	Gi0/0/0/3	10.0.1.13/30	10.0.1.14/30	ge-0/0/0	WH
BJ	Gi0/0/0/4	10.0.1.17/30	10.0.1.18/30	ge-0/0/0	XA
SH	Gi0/0/0/1	10.0.1.21/30	10.0.1.22/30	Gi0/0/0/0	GZ
SH	Gi0/0/0/2	10.0.1.25/30	10.0.1.26/30	ge-0/0/0	HZ
SH	Gi0/0/0/3	10.0.1.29/30	10.0.1.30/30	ge-0/0/0	NJ
SH	Gi0/0/0/4	10.0.1.33/30	10.0.1.34/30	ge-0/0/1	WH
SH	Gi0/0/0/10	10.0.3.1/30	10.0.3.2/30	e1	MGT_SRV
GZ	Gi0/0/0/2	10.0.1.37/30	10.0.1.38/30	ge-0/0/1	HZ
GZ	Gi0/0/0/3	10.0.1.41/30	10.0.1.42/30	ge-0/0/1	XA

我们使用的 Python 版本为 3.5.2。在安装完 Linux 的基本系统后，我们可以通过如下命令来获得需要的软件。

```
root@ubuntu:~# apt-get install python3 python3-pip
root@ubuntu:~# apt-get install python3-venv
root@ubuntu:~# apt-get install graphviz graphviz-dev
```

说明如下。

- ❑ python3 是 Python 3 解释器。
- ❑ python3-pip 是 Python 3 环境中的 pip 工具。
- ❑ python3-venv 是在 Python 3 中的虚拟环境创建模块。
- ❑ graphviz(<http://graphviz.org>) 是一个用于画图的工具。
- ❑ graphviz-dev 是 graphviz 的开发库，用于其他软件基于其二次开发的编译过程。

## 2. 创建 Python 3 的虚拟环境

命令如下：

```
root@ubuntu:~# export LC_ALL=C
root@ubuntu:~# python3 -m venv py3
root@ubuntu:~# source ./py3/bin/activate
```

## 3. 安装本案例需要的 Python 模块

命令如下：

```
(py3) root@ubuntu:~# pip install --upgrade pip
(py3) root@ubuntu:~# pip install pexpect
```

```
(py3) root@ubuntu:~# pip install ncclient
(py3) root@ubuntu:~# pip install exabgp
(py3) root@ubuntu:~# pip install networkx
(py3) root@ubuntu:~# pip install textfsm
(py3) root@ubuntu:~# pip install flask
(py3) root@ubuntu:~# pip install pyaml
```

说明如下。

- ❑ pexpect 用于模拟登录网络设备（在第 13 章已经使用过）。
- ❑ ncclient 用于通过 netconf 连接设备（在第 10 章已经使用过）。
- ❑ exabgp 用于处理 BGP 协议的模块。
- ❑ networkx 用于处理网络的拓扑计算的模块（在第 12 章已经使用过）。
- ❑ textfsm 用于处理半结构化文本的模块（在第 11 章已经使用过）。
- ❑ flask 为轻量级的 Web 框架，用于生成 HTTP API。
- ❑ pyaml 为处理 yaml 文件的模块。

安装完成后，我们可以使用如下命令来查看安装的模块。

```
(py3) root@ubuntu:~# pip freeze
asn1crypto==0.24.0
bcrypt==3.1.4
certifi==2017.11.5
cffi==1.11.2
chardet==3.0.4
click==6.7
cryptography==2.1.4
decorator==4.1.2
exabgp==4.0.2
Flask==0.12.2
idna==2.6
itsdangerous==0.24
Jinja2==2.10
lxml==4.1.1
MarkupSafe==1.0
ncclient==0.5.3
networkx==2.0
paramiko==2.4.0
pexpect==4.3.1
pkg-resources==0.0.0
ptyprocess==0.5.2
pyaml==17.12.1
pyasn1==0.4.2
pycparser==2.18
PyNaCl==1.2.1
PyYAML==3.12
six==1.11.0
textfsm==0.3.2
urllib3==1.22
Werkzeug==0.14.1
```



在安装一些模块的时候，由于它们会依赖一些模块，这些被依赖的模块会自己安装起来，因此，我们看到已经安装的模块会多于上面我们指定的模块。

## 14.2 网络拓扑的获取与分析

获取网络拓扑信息是进行网络分析的基础。在这个部分，我们将实现如何获取网络的拓扑信息。这里获取的网络拓扑包含物理拓扑和 IGP 协议拓扑两个部分。这两个拓扑的获取中，我们分别使用 SSH 登录设备和 NETCONF 登录设备。

### 14.2.1 物理拓扑的获取

众所周知，LLDP（Link Layer Discovery Protocol，链路层发现协议）是一种数据链路层协议，它通过在链路层上发送 LLDPDU（Link Layer Discovery Protocol Data Unit）来向其他设备通告自己的状态。我们可以使用这个协议来发现对端是什么设备，通过什么接口进行互联。

在这个实验中，我们可以登录到每一台设备上获取设备上的 LLDP 邻居关系信息，然后把这些信息整合在一起就是我们的物理拓扑信息了。

我们将使用第 13 章的方法来登录网络设备（用 pexpect 代用 SSH 工具登录设备）。

下面创建自己的一个模块用于处理和设备交互的工作。

```
# tree NetDevices/
NetDevices/
|-- Device.py
|-- EOS.py << 这个文件在本实验中并不使用
|-- IOS.py << 这个文件在本实验中并不使用
|-- IOSXR.py << 新增加的文件
|-- JUNOS.py << 修改第 13 章的内容
|-- NXOS.py << 这个文件在本实验中并不使用
`-- __init__.py
```

下面给出新增加的文件 IOSXR.py 的内容。

```
# cat NetDevices/IOSXR.py
#coding:utf-8
from NetDevices.Device import Device
from pexpect import EOF, TIMEOUT

class IOSXR(Device):

    def __init__(self, device):
        super(IOSXR, self).__init__(device)
        self.prompt = "RP/0/0/CPU0:" + self.hostname + "[>|#]\s?"

    async def login(self, prompt=""):
```

```

if not prompt:
    prompt = self.prompt
    await super(IXOSXR, self).login(prompt)
    await self._set_terminal_length_zero()

async def _set_terminal_length_zero(self):
    self.c.sendline("terminal length 0")
    try:
        i = await self.c.expect(self.prompt, async_=True)
    except EOF:
        pass
    except TIMEOUT:
        print("session timeout")

async def send_command(self, cmd=""):
    self.expect_list = []
    self.expect_list.append(self.prompt)
    result = []
    self.c.sendline(cmd)
    try:
        i = await self.c.expect(self.expect_list, timeout=5, async_=True)
        if i == 0:
            result.append(i)
            result.append((self.c.before + self.c.after).decode())
    except EOF:
        pass
    except TIMEOUT:
        print("session timeout")
    return result

async def get_config(self):
    return await self.send_command("show running-config")

async def get_lldp(self):
    return await self.send_command("show lldp neighbors")

```

这个文件的大部分内容和第 13 章提到的 NXOS.py 基本类似，只是修改了加粗和斜体部分的内容。对于获取设备的信息的操作，只是发送给设备的命令不一样。其他操作都是类似的。因此，这里把向设备发送命令的方法写为一个单独的方法。这样，当要完成类似功能的时候，我们就可以使用这个方法。

现在我们使用这个自己完成的模块来获取一下设备的 LLDP 信息。获取 LLDP 信息的代码如下：

```

$ cat lldp.py
#!/usr/bin/env python
#coding: utf-8
import asyncio
import yaml
import sys

```

```

from NetDevices import DeviceHandler

async def get_11dp(device):
    conn = DeviceHandler(device)
    conn.connect()
    await conn.login()
    r = await conn.get_11dp()
    for line in r[1].split("\r\n"):
        print(line)

deviceinfos = {}
try:
    yaml_cfg = sys.argv[1]
    # f = open(yaml_cfg)
    # deviceinfos = yaml.load(f.read())

except IndexError:
    print("please give yaml configure file")
    sys.exit(1)

f = open(yaml_cfg)
deviceinfos = yaml.load(f.read())

loop = asyncio.get_event_loop()
tasks = []
for device in deviceinfos.get("devices"):
    tasks.append(loop.create_task(get_11dp(device)))
loop.run_until_complete(asyncio.wait(tasks))

```

这段代码和第 13 章中通过异步并发的方式获取设备配置信息的内容几乎是一致的。这里我们只修改了加粗和斜体部分的内容。

下面是关于设备信息的 yaml 文件内容。

```

$ cat devices.yaml
devices:
  - BJ:
    hostname: BJ
    mgt_ip: 10.0.0.1
    vendor: Cisco
    OS_type: IOSXR
    username: admin
    password: admin
  - SH:
    hostname: SH
    mgt_ip: 10.0.0.2
    vendor: Cisco
    OS_type: IOSXR
    username: admin
    password: admin
  - GZ:

```



```

hostname: GZ
mgt_ip: 10.0.0.3
vendor: Cisco
OS_type: IOSXR
username: admin
password: admin
- HZ:
hostname: HZ
mgt_ip: 10.0.0.4
vendor: Juniper
OS_type: JUNOS
username: lab
password: lab123
- NJ:
hostname: NJ
mgt_ip: 10.0.0.5
vendor: Juniper
OS_type: JUNOS
username: lab
password: lab123
- WH:
hostname: WH
mgt_ip: 10.0.0.6
vendor: Juniper
OS_type: JUNOS
username: lab
password: lab123
- XA:
hostname: XA
mgt_ip: 10.0.0.7
vendor: Juniper
OS_type: JUNOS
username: lab
password: lab123

```

这个文件结构和第 13 章的内容也是非常类似的。

下面我们来运行一下上面的代码。

```

$ python lldp.py devices.yaml
show lldp neighbors| no-more

```

Local Interface	Parent Interface	Chassis Id	Port info
ge-0/0/1	-	02:21:a0:59:a4:06	Gi0/0/0/2
GZ.netdevops.cn			
ge-0/0/0	-	02:83:06:6a:a4:01	Gi0/0/0/2
SH.netdevops.cn			

```

lab@HZ>
show lldp neighbors| no-more

```

Local Interface	Parent Interface	Chassis Id	Port info
System Name			

```

ge-0/0/1          -          02:21:a0:59:a4:06   Gi0/0/0/2
    BJ.netdevops.cn
ge-0/0/0          -          02:83:06:6a:a4:01   Gi0/0/0/3
    SH.netdevops.cn

```

lab@NJ>

```

    show lldp neighbors| no-more
Local Interface      Parent Interface      Chassis Id      Port info
System Name
ge-0/0/0            -          02:21:a0:59:a4:06   Gi0/0/0/3
    BJ.netdevops.cn
ge-0/0/1            -          02:83:06:6a:a4:01   Gi0/0/0/4
    SH.netdevops.cn

```

lab@WH>

```

    show lldp neighbors| no-more
Local Interface      Parent Interface      Chassis Id      Port info
System Name
ge-0/0/1            -          02:21:a0:59:a4:06   Gi0/0/0/3
    GZ.netdevops.cn
ge-0/0/0            -          02:21:a0:59:a4:06   Gi0/0/0/4
    BJ.netdevops.cn

```

lab@XA>

```

show lldp neighbors
Mon Jan 1 14:59:13.391 UTC
Capability codes:

```

(R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device  
(W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID	Local Intf	Hold-time	Capability	Port ID
SH.netdevops.cn	Gi0/0/0/0	120	R	Gi0/0/0/0
GZ.netdevops.cn	Gi0/0/0/1	120	R	Gi0/0/0/1
NJ	Gi0/0/0/2	120	B,R	ge-0/0/1
WH	Gi0/0/0/3	120	B,R	ge-0/0/0
XA	Gi0/0/0/4	120	B,R	ge-0/0/0

Total entries displayed: 5

RP/0/0/CPU0:BJ#

```

show lldp neighbors
Mon Jan 1 14:59:14.740 UTC
Capability codes:

```

(R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device  
(W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID	Local Intf	Hold-time	Capability	Port ID
SH.netdevops.cn	Gi0/0/0/0	120	R	Gi0/0/0/1
BJ.netdevops.cn	Gi0/0/0/1	120	R	Gi0/0/0/1
HZ	Gi0/0/0/2	120	B,R	ge-0/0/1
XA	Gi0/0/0/3	120	B,R	ge-0/0/1

Total entries displayed: 4

RP/0/0/CPU0:GZ#

show lldp neighbors

Mon Jan 1 14:59:15.071 UTC

Capability codes:

(R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device

(W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID	Local Intf	Hold-time	Capability	Port ID
BJ.netdevops.cn	Gi0/0/0/0	120	R	Gi0/0/0/0
HZ	Gi0/0/0/2	120	B,R	ge-0/0/0
NJ	Gi0/0/0/3	120	B,R	ge-0/0/0
WH	Gi0/0/0/4	120	B,R	ge-0/0/1

Total entries displayed: 4

RP/0/0/CPU0:SH#

到这里，我们只是输出了设备的原始信息，并没有对这些信息进行处理。接下来，我们将使用第 11 章提到的 `textfsm` 模块对 LLDP 信息进行处理。我们对刚才的 `lldp.py` 代码又做了一些修改。这次修改的内容还以加粗和斜体的方式进行标注。

```
$ cat lldp.py
#!/usr/bin/env python
#coding: utf-8
import asyncio
import yaml
import sys
from NetDevices import DeviceHandler
import textfsm

def parser_lldp(device, lldp_result):
    OS_type = device.get("OS_type")
    # 定义 textfsm 模板的位置
    tmp_file = "./textfsm_tmp/lldp/%s.tmp" % OS_type
    fsm = textfsm.TextFSM(open(tmp_file))
    fsm_results = fsm.ParseText(lldp_result)
    return fsm_results

async def get_lldp(device):
    conn = DeviceHandler(device)
    conn.connect()
    await conn.login()
    r = await conn.get_lldp()
    lldp_lists = parser_lldp(device, r[1])
    print(lldp_lists)
```

< 略 > # 和上一段代码相同

下面给出 JUNOS 和 IOSXR 两个模板的内容。



## 1) JUNOS.tmp 模板的内容:

```
$ cat textfsm_tmp/lldp/JUNOS.tmp
Value device_id (\S+)
Value local_intf (\w+\-[\|\\\d]{5})
Value remote_intf (\S+)

Start
^Local Interface      Parent Interface      Chassis Id          Port info
  System Name
^${local_intf}\s+\S+\s+\S+\s+${remote_intf}\s+${device_id} -> Record

EOF
```

## 2) IOSXR.tmp 模板的内容:

```
$ cat textfsm_tmp/lldp/IOSXR.tmp
Value device_id (\S+)
Value local_intf (\w+[\|\\\d]{7})
Value remote_intf (\S+)

Start
^Device ID           Local Intf           Hold-time Capability  Port ID
^${device_id}\s+${local_intf}\s+\d+\s+\S+\s+${remote_intf} -> Record

EOF
```

运行结果如下:

```
$ python lldp.py devices.yaml
[['BJ.netdevops.cn', 'ge-0/0/0', 'Gi0/0/0/3'], ['SH.netdevops.cn', 'ge-0/0/1', 'Gi0/0/0/4']]
[['BJ.netdevops.cn', 'ge-0/0/1', 'Gi0/0/0/2'], ['SH.netdevops.cn', 'ge-0/0/0', 'Gi0/0/0/3']]
[['GZ.netdevops.cn', 'ge-0/0/1', 'Gi0/0/0/2'], ['SH.netdevops.cn', 'ge-0/0/0', 'Gi0/0/0/2']]
[['GZ.netdevops.cn', 'ge-0/0/1', 'Gi0/0/0/3'], ['BJ.netdevops.cn', 'ge-0/0/0', 'Gi0/0/0/4']]
[['BJ.netdevops.cn', 'Gi0/0/0/0', 'Gi0/0/0/0'], ['HZ', 'Gi0/0/0/2', 'ge-0/0/0'], ['NJ', 'Gi0/0/0/3', 'ge-0/0/0'], ['WH', 'Gi0/0/0/4', 'ge-0/0/1']]
[['SH.netdevops.cn', 'Gi0/0/0/0', 'Gi0/0/0/0'], ['GZ.netdevops.cn', 'Gi0/0/0/1', 'Gi0/0/0/1'], ['NJ', 'Gi0/0/0/2', 'ge-0/0/1'], ['WH', 'Gi0/0/0/3', 'ge-0/0/0'], ['XA', 'Gi0/0/0/4', 'ge-0/0/0']]
[['SH.netdevops.cn', 'Gi0/0/0/0', 'Gi0/0/0/1'], ['BJ.netdevops.cn', 'Gi0/0/0/1', 'Gi0/0/0/1'], ['HZ', 'Gi0/0/0/2', 'ge-0/0/1'], ['XA', 'Gi0/0/0/3', 'ge-0/0/1']]
```

现在我们已经获取了所有网络设备上的 LLDP 邻居信息,并且已经进行了格式化处理,但是,这里我们还没有使用 networkx 模块加载这些数据。读者可以尝试完成用 networkx 加载拓扑的部分。

### 14.2.2 ISIS 协议拓扑的获取

除了物理拓扑之外，网络的 IGP 协议的拓扑也是非常重要的。不过 IGP 拓扑的获取并不像 LLDP 一样，我们不需要从每一台设备上获取信息，而只需要登录到部分设备就可以了（对于 OSPF，一个 area 只需要一台设备；对于 ISIS，一个 level area，只需要一台设备）。

在这个实验中，所有的 ISIS 路由器都运行在 Level 2 中。因此我们只需要登录其中一台设备就可以获取全网的拓扑了。这次我们使用 netconf 协议登录到其中一台 JUNOS 设备上来获取网络的 ISIS 协议拓扑。其代码如下：

```
#!/usr/bin/env python
from pprint import pprint
from ncclient import manager

# 设备的信息
device = {"host": "10.0.0.4",
          "port": 830,
          "username": "lab",
          "password": "lab123",
          "hostkey_verify": False,
          "device_params": {"name": "junos"}}

nc = manager.connect(**device)
# 通过 netconf rpc 获取 ISIS 的 XML 信息
isis_xml = nc.rpc("""<get-isis-database-information>
                    <detail/>
                  </get-isis-database-information>""")

topology = {}
# 以下代码用于解析 XML 信息
isis_entry = "//isis-database/level[text()='2']/following-sibling::*"
for isis_database_entry in isis_xml.xpath(isis_entry):
    lsp_id = isis_database_entry.xpath("lsp-id")
    lsp_id = lsp_id[0].text
    lsp_id = lsp_id.replace(".00-00", "")
    topology[lsp_id] = {"nodes": [], "address_prefix": []}

    for isis_neighbor in isis_database_entry.xpath("isis-neighbor"):
        isis_neighbor_id = isis_neighbor.xpath("is-neighbor-id")
        isis_neighbor_id = isis_neighbor_id[0].text
        isis_neighbor_id = isis_neighbor_id.replace(".00", "")

        isis_neighbor_metric = isis_neighbor.xpath("metric")
        isis_neighbor_metric = isis_neighbor_metric[0]
        isis_neighbor_metric = isis_neighbor_metric.text
        isis_neighbor_metric = int(isis_neighbor_metric)
        node_metric = (isis_neighbor_id, isis_neighbor_metric)
        topology[lsp_id]["nodes"].append(node_metric)
```

```

for isis_prefix in isis_database_entry.xpath("isis-prefix"):
    address_prefix = isis_prefix.xpath("address-prefix")[0].text
    addr_metric = isis_prefix.xpath("metric")
    addr_metric = addr_metric[0].text
    addr_metric = int(addr_metric)
    prefix_metric = (address_prefix, addr_metric)
    topology[lsp_id]["address_prefix"].append(prefix_metric)

```

```
pprint(topology)
```

运行结果如下：

```

$ python isis.py
{'BJ': {'address_prefix': [('10.0.0.1/32', 10),
                           ('10.0.1.0/30', 10),
                           ('10.0.1.4/30', 10),
                           ('10.0.1.8/30', 10),
                           ('10.0.1.12/30', 10),
                           ('10.0.1.16/30', 10)],
        'nodes': [('SH', 10), ('GZ', 10), ('NJ', 10), ('WH', 10), ('XA', 10)]},
'GZ': {'address_prefix': [('10.0.0.3/32', 10),
                           ('10.0.1.4/30', 10),
                           ('10.0.1.20/30', 10),
                           ('10.0.1.36/30', 10),
                           ('10.0.1.40/30', 10)],
        'nodes': [('BJ', 10), ('SH', 10), ('HZ', 10), ('XA', 10)]},
'HZ': {'address_prefix': [('10.0.0.4/32', 0),
                           ('10.0.1.24/30', 10),
                           ('10.0.1.36/30', 10)],
        'nodes': [('SH', 10), ('GZ', 10)]},
'NJ': {'address_prefix': [('10.0.0.5/32', 0),
                           ('10.0.1.8/30', 10),
                           ('10.0.1.28/30', 10)],
        'nodes': [('BJ', 10), ('SH', 10)]},
'SH': {'address_prefix': [('10.0.0.2/32', 10),
                           ('10.0.1.0/30', 10),
                           ('10.0.1.20/30', 10),
                           ('10.0.1.24/30', 10),
                           ('10.0.1.28/30', 10),
                           ('10.0.1.32/30', 10),
                           ('10.0.3.0/30', 0)],
        'nodes': [('BJ', 10), ('GZ', 10), ('HZ', 10), ('NJ', 10), ('WH', 10)]},
'WH': {'address_prefix': [('10.0.0.6/32', 0),
                           ('10.0.1.12/30', 10),
                           ('10.0.1.32/30', 10)],
        'nodes': [('BJ', 10), ('SH', 10)]},
'XA': {'address_prefix': [('10.0.0.7/32', 0),
                           ('10.0.1.16/30', 10),
                           ('10.0.1.40/30', 10)],
        'nodes': [('BJ', 10), ('GZ', 10)]}}

```



### 14.2.3 网络拓扑的路径分析

现在，我们来模拟一下 ISIS 是如何计算路由的。在 ISIS 协议中，我们先计算两个 node 之间的最短路径 metric，然后加上所有接口地址的 metric。

代码如下（这段代码是上面从 ISIS 获取拓扑信息的后续）：

```
import networkx as nx

# 创建一个有向图
G = nx.DiGraph()

# 把 14.2.2 节中的结果放在 networkx 中的有向图中。使用 add_edge 方法添加拓扑中的边就可以了
for node, infos in topology.items():
    for remote_node in infos.get("nodes"):
        G.add_edge(node, remote_node[0], weight=remote_node[1])

# 获取所有的 node 节点
nodes = G.nodes

def get_local_prefix(node):
    prefixes = []
    for prefix in topology[node]["address_prefix"]:
        prefixes.append(prefix[0])
    return prefixes

for node in nodes:
    print("router %s 's route tables:" % node)
    print("%-18s %-7s %-7s" % ("prefix", "nexthop", "metric"))
    print("-"*30)
    for remote_node in nodes:
        if remote_node == node:
            for prefix in topology.get(node).get("address_prefix"):
                print("%-18s %-7s %-7s" % (prefix[0], "Local", prefix[1]))
        else:
            # 使用 dijkstra 算法计算两个节点之间的最短路径
            path = nx.dijkstra_path(G, node, remote_node, weight="weight")
            # 使用 dijkstra 算法计算最短路径 metric (即长度)
            path_length = nx.dijkstra_path_length(G, node, remote_node, weight="weight")

            for prefix in topology.get(remote_node).get("address_prefix"):
                if prefix[0] not in get_local_prefix(node):
                    print("%-18s %-7s %-7s" % (prefix[0], path[1], path_length +
                                                prefix[1]))
    print("-"*30)
```

运行这个代码可以输出所有路由器上的所有 IGP 路由信息。

```
$ python isis.py
router BJ 's route tables:
```

prefix	nextthop	metric
10.0.0.1/32	Local	10
10.0.1.0/30	Local	10
10.0.1.4/30	Local	10
10.0.1.8/30	Local	10
10.0.1.12/30	Local	10
10.0.1.16/30	Local	10
10.0.0.4/32	GZ	20
10.0.1.24/30	GZ	30
10.0.1.36/30	GZ	30
10.0.0.5/32	NJ	10
10.0.1.28/30	NJ	20
10.0.0.3/32	GZ	20
10.0.1.20/30	GZ	20
10.0.1.36/30	GZ	20
10.0.1.40/30	GZ	20
10.0.0.6/32	WH	10
10.0.1.32/30	WH	20
10.0.0.7/32	XA	10
10.0.1.40/30	XA	20
10.0.0.2/32	SH	20
10.0.1.20/30	SH	20
10.0.1.24/30	SH	20
10.0.1.28/30	SH	20
10.0.1.32/30	SH	20
10.0.3.0/30	SH	10

=====

< 其他设备略 >

由于 networkx 实现的 dijkstra 算法并不能给出所有的等价路径，因此，输出结果中可能会有一些路径的缺失。如果需要计算所有的等价路径，我们需要使用 networkx 中的 all\_shortest\_paths 方法。关于这个方法的使用，读者可以尝试修改上面的代码来实现多路径的等价路径。

### 14.3 网络流量工程应用

网络流量工程的广义定义是指：基于不同的业务流量特性，选取特定的传输路径的处理过程。流量工程的实施过程可以分为两个部分。

第一部分是对路径的选择，我们是基于最短路径的选择，还是基于非最短路径的选择，又或者是基于经过特定点（特定中间节点或特定链路）的选择（基于最短路径的变形而已）。

第二部分是如何把选择好的路径应用在网络设备中。根据第 2 章提到的内容，我们可以通过三种方法来实现它：①通过修改网络设备配置来实现；②通过修改转发路径的转发

表（可能是路由表，也可能是 MPLS 标签或者其他形式）来实现；③通过在数据包中加入特定的包头标识（这个标识可以是 MPLS 标签，也可以是 IPv6 的包头，又或者是其他信息，而添加包头的位置可以是网络的边缘节点，也可以是其他节点）来实现。

通常我们在选择转发路径时都基于最短路径，这里的最短可以是链路成本最小，可以是延迟最小，还可以是链路抖动最低等。我们都有使用导航软件的经验，导航软件通常会给我们推荐几个方案，有时间最短的方案，有路程最短的方案，还有红绿灯最少的方案等。这些方案都是基于最短路径方法进行的路径计算。在 IP 网络中，我们同样可以实现类似的方案。在现存的 IGP 协议中，每条链路的 cost（或者是 metric，即代价）是固定的。它们通常是基于链路的物理带宽或者是基于传输的距离（传输距离是链路成本的映射和链路延迟的反应），也可能是根据规划者的经验（这个经验值可以来自前期对网络的仿真计算）。这些值一旦被确定后，在网络的长期运行过程中是不会被频繁（这里说的频繁指其稳定期通常以年为单位）修改的。

在本节中，我们将提供一个案例的简单实现（只完成基本功能），即基于传输距离和使用 BGP 下发路由前缀的方式实现流量工程。

14.3.1 基本信息

首先，我们给出本案例的一些基本信息。表 14-3 是设备之间的传输距离（这个距离只是一个粗略的假设值，并不等于真实环境中的值）。

表 14-3 传输距离

A 端设备名	Z 端设备名	距离（千米）	A 端设备名	Z 端设备名	距离（千米）
BJ	SH	1250	SH	HZ	200
BJ	GZ	2200	SH	NJ	320
BJ	NJ	1050	SH	WH	850
BJ	WH	1200	GZ	HZ	1300
BJ	XA	1100	GZ	XA	1700
SH	GZ	1500			

表 14-3 的内容使用 YAML 文件可表示为

```
$ cat distance.yaml
BJ:
  SH: 1250
  GZ: 2200
  NJ: 1050
  WH: 1200
  XA: 1100
SH:
  GZ: 1500
```



```

HZ: 200
NJ: 320
WH: 850
GZ:
  HZ: 1300
  XA: 1700

```

其次, 我们给出这个案例的逻辑框图, 如图 14-2 所示。逻辑框图分为两大部分, 第一部分用于完成路径的计算, 第二部分使用 BGP 协议对路径经过的网络设备下发路由前缀。在这个拓扑中, ISIS 作为 IGP 协议, 通过 ISIS 协议可以获得 IP 网络拓扑。

### 14.3.2 路径计算

下面给出第一部分的代码, 其基本功能是获取路由数据。

```

#!/usr/bin/env python
#coding: utf-8
import sys
from ncclient import manager
import networkx as nx
from networkx.utils.misc import pairwise
import yaml
from pprint import pprint

# 命令行输入参数的获取
def usage():
    try:
        source = sys.argv[1]
        destination = sys.argv[2]
        prefix = sys.argv[3]
        return(source, destination, prefix)
    except IndexError:
        print("%s [source] [destination] [prefix] " %sys.argv[0])

# 通过 ISIS 数据库获取 IP 网络拓扑信息。这部分的代码基本和 14.2.2 节一致。只是把 14.2.2 节的代码改
# 成了一个函数
def get_isis_topology(device):
    nc = manager.connect(**device)
    isis_xml = nc.rpc("""<get-isis-database-information>
                        <detail/>
                    </get-isis-database-information>""")
    topology = {}
    isis_entry = "//isis-database/level[text()='2']/following-sibling::*"
    for isis_database_entry in isis_xml.xpath(isis_entry):
        lsp_id = isis_database_entry.xpath("lsp-id")
        lsp_id = isis_database_entry[0].text
        lsp_id = lsp_id.replace(".00-00","")

```

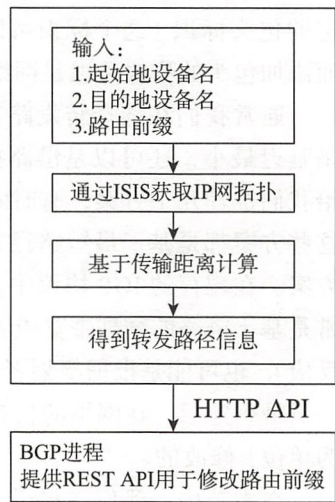


图 14-2 逻辑框图

```

topology[lsp_id] = {"nodes": [], "address_prefix": []}

for isis_neighbor in isis_database_entry.xpath("isis-neighbor"):
    isis_neighbor_id = isis_neighbor.xpath("is-neighbor-id")
    isis_neighbor_id = isis_neighbor_id[0].text
    isis_neighbor_id = isis_neighbor_id.replace(".00", "")

    isis_neighbor_metric = isis_neighbor.xpath("metric")
    isis_neighbor_metric = isis_neighbor_metric[0]
    isis_neighbor_metric = isis_neighbor_metric.text
    isis_neighbor_metric = int(isis_neighbor_metric)
    node_metric = (isis_neighbor_id, isis_neighbor_metric)
    topology[lsp_id]["nodes"].append(node_metric)

for isis_prefix in isis_database_entry.xpath("isis-prefix"):
    address_prefix = isis_prefix.xpath("address-prefix")[0].text
    addr_metric = isis_prefix.xpath("metric")
    addr_metric = addr_metric[0].text
    addr_metric = int(addr_metric)
    prefix_metric = (address_prefix, addr_metric)
    topology[lsp_id]["address_prefix"].append(prefix_metric)

```

```

G = nx.Graph()
for node, infos in topology.items():
    for remote_node in infos.get("nodes"):
        G.add_edge(node, remote_node[0], weight=remote_node[1])
return G

```

# 从 yaml 配置文件中获取链路之间的距离  
# 在 ISIS 配置中, 所有链路的 metric 都是默认值, 并没有使用链路的距离作为 metric  
# 这里获得的值将用于更新 ISIS 拓扑中的 metric 值

```
def load_distance_from_yamlfile(yaml_file):
```

```

    try:
        yaml_infos = yaml.load(open(yaml_file).read())
        edges = {}
        for node, edge in yaml_infos.items():
            for remote_node, weight in edge.items():
                edges[(node, remote_node)] = int(weight)
        return edges
    except FileNotFoundError:
        print("%s can't find" %yaml_file)
        sys.exit(1)

```

# 读取每台设备的 loopback 地址信息  
# 这个信息将用于路由表的生成, 作为路由表中的下一跳信息使用

```
def load_loopback_from_yamlfile(yaml_file):
```

```

    try:
        nodes = yaml.load(open(yaml_file).read())
        return nodes
    except FileNotFoundError:
        print("%s can't find" %yaml_file)

```

```

sys.exit(1)

# 用于修改 ISIS 拓扑中的 metric 值。将所有的值改成基于链路距离的拓扑
def load_distance(G, distance):
    for edge in G.edges:
        if distance.get((edge[0],edge[1])):
            G.edges[edge[0],edge[1]]["weight"] = distance.get((edge[0],edge[1]))
        else:
            G.edges[edge[0],edge[1]]["weight"] = distance.get((edge[1],edge[0]))
    return G

# 计算源节点和目的节点之间的最短路径 (如果有多条路径只会选择一条), 并产生路径沿途的路由条目
def get_route_entries(G, source, target, prefix):
    path = nx.shortest_path(G, source, target, weight="weight")
    entries = {}
    for edge in pairwise(path):
        entries[edge[0]] = {"prefix": prefix, "next-hop": G.nodes[edge[1]]["loop"]}

    return entries

if __name__ == "__main__":

    args = usage()
    if not args:
        sys.exit(1)

    isis_device = {"host": "10.0.0.4",
                   "port": 830,
                   "username": "lab",
                   "password": "lab123",
                   "hostkey_verify": False,
                   "device_params": {"name": "junos"}}

    # 获取 ISIS 拓扑
    t = get_isis_topology(isis_device)
    # 加载链路的距离数据为链路的 metric
    distance = load_distance_from_yamlfile("./distance.yaml")
    t = load_distance(t,distance)
    # 添加每台设备的 loopback 接口地址的数据
    loop_attr = load_loopback_from_yamlfile("./devices_loopinf.yaml")
    for node in t.nodes:
        t.nodes[node]["loop"] = loop_attr[node]
    # 获取源节点和目的节点之间的最短路径已经生成沿途路径下的所有路由条目
    entries = get_route_entries(t, *args)
    pprint(entries)

```

### 14.3.3 BGP 服务

在第二部分中, 我们使用 `exabgp` 作为 BGP 的服务进程。首先, 我们需要用 `exabgp` 和所有的设备建立 BGP 邻居关系。`exabgp` 的配置文件如下 (省略网络设备侧的配置):



```

# Well-Known communities
# no-export - 65535:65281 # do not advertise to any eBGP peers
# no-advertise - 65535:65282 # do not advertise to any BGP peer
# local-as - 65535:65283 # do not advertise to peers outside the local as

template {
    neighbor ibgp {
        router-id 10.0.3.2;
        local-address 10.0.3.2;
        local-as 4000;
        peer-as 4000;
        hold-time 180;
        static {
            route 1.0.0.0/8 next-hop 10.0.3.2 community 65535:65282;
        }
    }
}

neighbor 10.0.0.1 {
    inherit ibgp;
    description "to BJ BGP peer";
}

neighbor 10.0.0.2 {
    inherit ibgp;
    description "to SH BGP peer";
}

neighbor 10.0.0.3 {
    inherit ibgp;
    description "to GZ BGP peer";
}

neighbor 10.0.0.4 {
    inherit ibgp;
    description "to HZ BGP peer";
}

neighbor 10.0.0.5 {
    inherit ibgp;
    description "to NJ BGP peer";
}

neighbor 10.0.0.6 {
    inherit ibgp;
    description "to WH BGP peer";
}

neighbor 10.0.0.7 {
    inherit ibgp;

```

```
description "to XA BGP peer";
}
```

现在先运行一下 `exabgp`，然后在网络设备上检查 `1.0.0.0/8` 的路由信息。如果通过 `root` 登录服务器，需要指定 `exabgp` 运行的用户为 `root`（当然也可以通过修改环境变量的配置来完成）。

```
# env exabgp.daemon.user=root exabgp exabgp.ini
```

在 `10.0.0.1` 与 `10.0.0.4` 上检查路由表的情况。

#### 1) 检查设备 `10.0.0.1` 的情况：

```
RP/0/0/CPU0:BJ#show route 1.0.0.0/8
```

```
Tue Jan  2 07:48:43.451 UTC
```

```
Routing entry for 1.0.0.0/8
```

```
Known via "bgp 4000", distance 200, metric 0, type internal
```

```
Installed Jan  2 07:48:13.881 for 00:03:30
```

```
Routing Descriptor Blocks
```

```
10.0.3.2, from 10.0.3.2
```

```
Route metric is 0
```

```
No advertising protos.
```

```
RP/0/0/CPU0:BJ#show bgp 1.0.0.0/8 detail
```

```
Tue Jan  2 07:51:46.829 UTC
```

```
BGP routing table entry for 1.0.0.0/8
```

```
Versions:
```

```
Process          bRIB/RIB  SendTblVer
```

```
Speaker          25        25
```

```
Flags: 0x04001001+0x00000000;
```

```
Last Modified: Jan  2 07:48:13.881 for 00:03:32
```

```
Paths: (1 available, best #1, not advertised to any peer)
```

```
Not advertised to any peer
```

```
Path #1: Received by speaker 0
```

```
Flags: 0x4000000001048007, import: 0x20
```

```
Not advertised to any peer
```

```
Local, (received & used)
```

```
10.0.3.2 (metric 10) from 10.0.3.2 (10.0.3.2)
```

```
Origin IGP, localpref 100, valid, internal, best, group-best
```

```
Received Path ID 0, Local Path ID 1, version 25
```

```
Community: no-advertise
```

#### 2) 检查设备 `10.0.0.4` 的情况：

```
lab@HZ> show route 1.0.0.0/8 detail
```

```
inet.0: 22 destinations, 22 routes (22 active, 0 holddown, 0 hidden)
```

```
1.0.0.0/8 (1 entry, 1 announced)
```

```
*BGP Preference: 170/-101
```

```
Next hop type: Indirect
```

```
Address: 0x940e770
```

```

Next-hop reference count: 3
Source: 10.0.3.2
Next hop type: Router, Next hop index: 521
Next hop: 10.0.1.25 via ge-0/0/0.0, selected
Session Id: 0x3
Protocol next hop: 10.0.3.2
Indirect next hop: 0x97cc000 1048574 INH Session ID: 0x8
State: <Active Int Ext>
Local AS: 4000 Peer AS: 4000
Age: 1:41 Metric2: 10
Validation State: unverified
Task: BGP_4000.10.0.3.2+46552
Announcement bits (2): 0-KRT 4-Resolve tree 4
AS path: I
Communities: no-advertise
Accepted
Localpref: 100
Router ID: 10.0.3.2

```

在输出的结果中，加粗和斜体的信息都是由 exabgp BGP 进程发出的。

下面我们通过 exabgp API 来实现把 exabgp API 转换为 REST API。这里我们需要在配置文件中指定一段 Python 代码。下面是修改后的 exabgp 配置文件。

```

process http_api {
    run ./http_api.py;
    encoder json;
}

template {
    neighbor ibgp {
        router-id 10.0.3.2;
        local-address 10.0.3.2;
        local-as 4000;
        peer-as 4000;
        hold-time 180;
        api {
            processes [ http_api ];
            neighbor-changes;
            send {
                packets;
            }
        }
        static {
            route 1.0.0.0/8 next-hop 10.0.3.2 community 65535:65282;
        }
    }
}
<... 后略 ...> # 同上一个文件

```

在这个文件中，加粗和斜体的部分是新增加的配置行。其含义是使用 exabgp API 与其



交互。这里 `run ./http_api.py` 指定了 Python 代码的位置。

下面是 `http_api.py` 的文件内容。

```
#!/usr/bin/env python

from flask import Flask, request
from sys import stdout

# 实例化一个 Flask
app = Flask(__name__)

# 定义一个函数和 URL 进行绑定
@app.route("/", methods=["POST"])
def command():
    # 接受 POST 传递来的内容
    command = request.form["command"]

    # 给发送的路由前缀添加一个 no-advertise 的公有属性
    if "65535:65282" not in command and "announce route" in command:
        command = command + " community 65535:65282"

    # 发送命令给 exabgp
    stdout.write("%s\n" % command)
    stdout.flush()

    # 返回刚才输入的命令内容
    return "%s\n" % command

if __name__ == "__main__":
    # 启动 Flask app 进程。默认的服务地址为 localhost，端口为 5000
    app.run()
```

重新启动 exabgp 的进程：

```
# env exabgp.daemon.user=root exabgp exabgp.ini
19:06:18 | 23217 | welcome | Thank you for using ExaBGP
19:06:18 | 23217 | version | 4.0.2-1c737d99
<略>
19:06:18 | 23217 | configuration | performing reload of exabgp 4.0.2-1c737d99
19:06:18 | 23217 | reactor | loaded new configuration successfully
19:06:18 | 23217 | reactor | connected to peer-7 with outgoing-1 10.0.3.2-10.0.0.7
19:06:18 | 23217 | reactor | connected to peer-4 with outgoing-3 10.0.3.2-10.0.0.4
19:06:18 | 23217 | reactor | connected to peer-3 with outgoing-4 10.0.3.2-10.0.0.3
19:06:18 | 23217 | reactor | connected to peer-5 with outgoing-5 10.0.3.2-10.0.0.5
19:06:18 | 23217 | reactor | connected to peer-6 with outgoing-6 10.0.3.2-10.0.0.6
19:06:18 | 23217 | reactor | connected to peer-1 with outgoing-7 10.0.3.2-10.0.0.1
```

```
19:06:18 | 23217 | reactor | connected to peer-2 with outgoing-2 10.0.3.2-10.0.0.2
```

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

最后，我们可以看到 Flask 的进程也启动了。这时我们可以在这台机器上用 HTTP API 进行调用，并添加一条路由信息。

```
curl --form "command=neighbor 10.0.0.2 announce route 202.100.1.0/24 next-hop 10.0.0.1" http://localhost:5000/
neighbor 10.0.0.2 announce route 202.100.1.0/24 next-hop 10.0.0.1 community 65535:65282
```

该命令使用 curl 来完成一个 HTTP 请求。在这个命令中，neighbor 10.0.0.2 指定了向哪个邻居发送路由更新信息。announce route 202.100.1.0/24 next-hop 10.0.0.1 表明需要通告一条什么样的路由信息。

现在我们在 10.0.0.2 这台设备上来查看这条通告的路由信息。由于我们只发送了一个 BGP 的邻居，并且路由信息中包含了 no-advertise 的公有属性，所以其他的设备上是没有这条路由信息的。

```
RP/0/0/CPU0:SH#show route 202.100.1.0/24
Tue Jan  2 11:19:22.375 UTC
```

```
Routing entry for 202.100.1.0/24
  Known via "bgp 4000", distance 200, metric 0, type internal
  Installed Jan  2 11:13:55.418 for 00:05:27
  Routing Descriptor Blocks
    10.0.0.1, from 10.0.3.2
      Route metric is 0
  No advertising protos.
```

```
RP/0/0/CPU0:SH#show bgp 202.100.1.0/24 detail
```

```
Tue Jan  2 11:19:34.705 UTC
```

```
BGP routing table entry for 202.100.1.0/24
```

```
Versions:
```

```
Process          bRIB/RIB  SendTblVer
Speaker          50        50
```

```
Flags: 0x04001001+0x00000000;
```

```
Last Modified: Jan  2 11:13:55.834 for 00:05:38
```

```
Paths: (1 available, best #1, not advertised to any peer)
```

```
Not advertised to any peer
```

```
Path #1: Received by speaker 0
```

```
Flags: 0x4000000001040207, import: 0x20
```

```
Not advertised to any peer
```

```
Local, (Received from a RR-client)
```

```
10.0.0.1 (metric 20) from 10.0.3.2 (10.0.3.2)
```

```
Origin IGP, localpref 100, valid, internal, best, group-best
```

```
Received Path ID 0, Local Path ID 1, version 50
```

```
Community: no-advertise
```

### 14.3.4 调用 BGP HTTP API

到目前为止，我们已经把两个基本模块都构建好了。现在我们需要把第一部分和第二部分结合起来。这里只需要修改第一部分的代码。我们只给出在第一部分中增加的代码。

# 对计算完成的路径完成对设备的下发部署

```
def call_http_api(entrys, http_host="localhost"):
    for entry in entrys[::-1]:
        neighbor = entry.get("neighbor")
        prefix = entry.get("prefix")
        next_hop = entry.get("next-hop")

        exabgp_cmd = "command=neighbor %s \
            announce route %s \
            next-hop %s" %(neighbor, prefix, next_hop)
        cmd = 'curl --form "%s" ' %(exabgp_cmd)
        cmd = cmd + " http://%s:5000/" %http_host
        os.system(cmd)

if __name__ == "__main__":
    argvs = usage()
    if not argvs:
        sys.exit(1)
    isis_device = {"host": "10.0.0.4",
        "port": 830,
        "username": "lab",
        "password": "lab123",
        "hostkey_verify": False,
        "device_params": {"name": "junos"}}

    t = get_isis_topology(isis_device)
    distance = load_distance_from_yamlfile("./distance.yaml")
    t = load_distance(t,distance)

    loop_attr = load_loopback_from_yamlfile("./devices_loopinf.yaml")
    for node in t.nodes:
        t.nodes[node]["loop"] = loop_attr[node]
    entrys = get_route_entrys(t, *argvs)
    call_http_api(entrys)
```

其中，加粗和斜体部分是新增加的内容。

### 14.3.5 结果测试

在这个测试拓扑中，BGP 协议并没有承载业务路由。我们希望通过路径计算，找到 XA 节点到 HZ 节点的转发路径，其路由前缀为 100.1.1.0/24。为了测试其流量的情况，我们在 HZ 的 ge-0/0/2 中添加一个 IP 地址 100.1.1.1/24。



### (1) 启动 exabgp 进程

```
# env exabgp.daemon.user=root exabgp exabgp.ini
```

### (2) 在 XA 节点 ping HZ 节点上的 100.1.1.1 地址

```
lab@XA> ping 100.1.1.1
PING 100.1.1.1 (100.1.1.1): 56 data bytes
ping: sendto: No route to host
ping: sendto: No route to host
```

### (3) 启动路径计算的程序，计算后下发 BGP 路由信息

```
# python path_compute.py XA HZ 100.1.1.0/24
neighbor 10.0.0.2 announce route 100.1.1.0/24 next-hop 10.0.0.4 community 65535:65282
neighbor 10.0.0.1 announce route 100.1.1.0/24 next-hop 10.0.0.2 community 65535:65282
neighbor 10.0.0.7 announce route 100.1.1.0/24 next-hop 10.0.0.1 community 65535:65282
```

### (4) 在 XA 节点 ping HZ 节点上的 100.1.1.1 地址

```
lab@XA> ping 100.1.1.1
PING 100.1.1.1 (100.1.1.1): 56 data bytes
ping: sendto: No route to host
ping: sendto: No route to host
64 bytes from 100.1.1.1: icmp_seq=7 ttl=62 time=7.913 ms
64 bytes from 100.1.1.1: icmp_seq=8 ttl=62 time=6.880 ms
64 bytes from 100.1.1.1: icmp_seq=9 ttl=62 time=6.832 ms
```

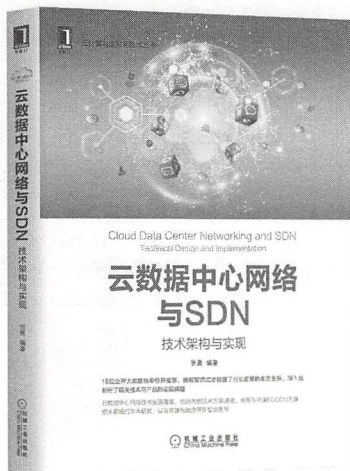
在这个案例中，我们首先使用 NETCONF 协议获取 ISIS 的数据库信息。然后，使用 net-workx 进行了路径计算，最后使用 exabgp 和 Flask 完成了 BGP 路由表的下发。这个例子并不太适合正式的生产环境，这种方式非常类似在每台设备上下发静态路由，并且当网络发生故障时会出现网络无法正常收敛的情况。但是，笔者只是希望通过这样一个例子让大家熟悉一下对网络拓扑的计算。在真实的网络环境中，为了实现流量工程要做的工作还有很多。

## 14.4 小结

相比第 13 章的例子，本章的这个例子其实更复杂，使用的 Python 模块和工具更多，例如，有处理文本的 TextFSM 模块，有用于路径计算的 networkx 模块，有用于 BGP 协议的 exabgp 工具，还有用于开发 HTTP API 的 Flask 模块等。另外，这个例子并不是仅限于操作网络的配置文件，而是直接在协议层面完成对网络转发路径的影响。

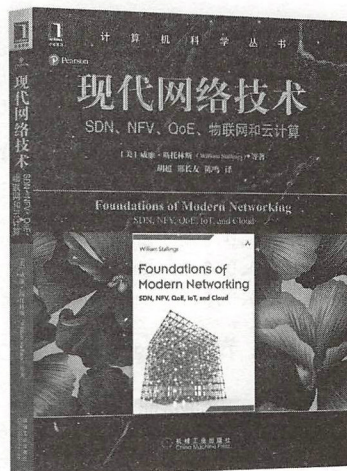
到这里，我们已经完成了本书的所有内容。希望本书的内容能为那些有志于向 NetDev-Ops 转型的传统网络工程师提供一点帮助。

## 推荐阅读



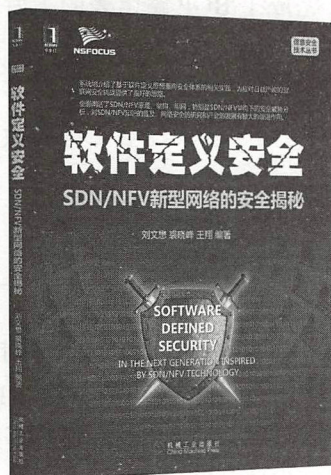
### 云数据中心网络与SDN：技术架构与实现

作者：张晨 编著 ISBN: 978-7-111-59121-4 定价：119.00元



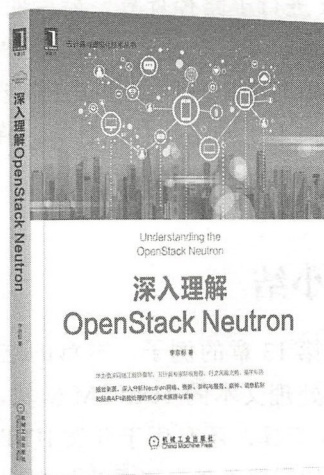
### 现代网络技术：SDN、NFV、QoE、物联网和云计算

作者：威廉·斯托林斯 等 ISBN: 978-7-111-58664-7 定价：99.00元



### 软件定义安全：SDN/NFV新型网络的安全揭秘

作者：刘文懋 裴晓峰 王翔 编著 ISBN: 978-7-111-54836-2 定价：59.00元

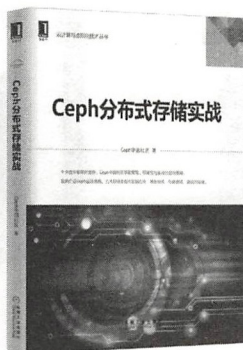
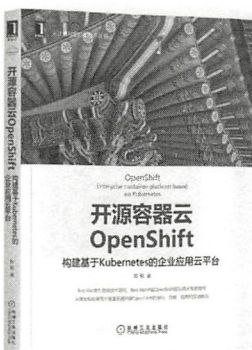
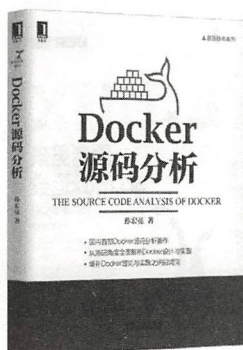
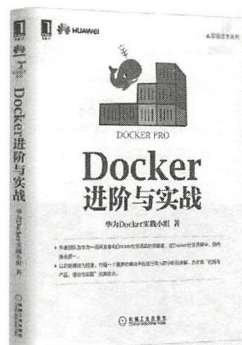
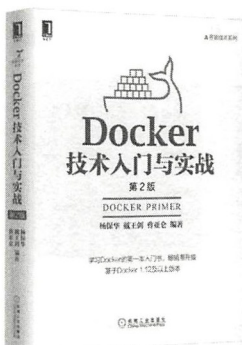
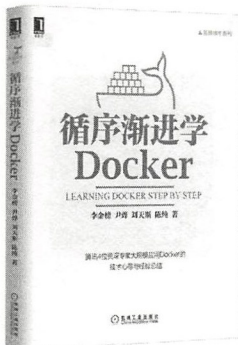


### 深入理解OpenStack Neutron

作者：李宗标 ISBN: 978-7-111-58448-3 定价：89.00元

# 推荐阅读

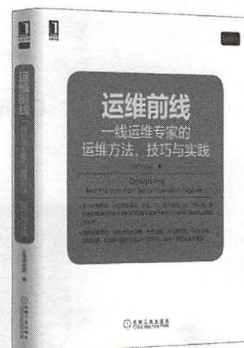
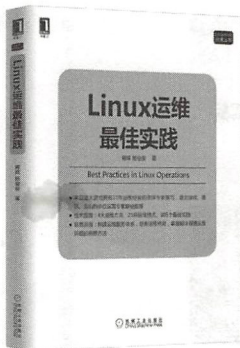
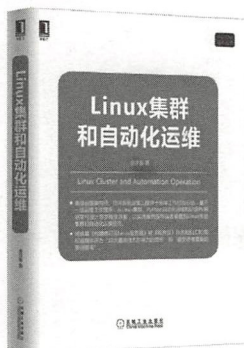
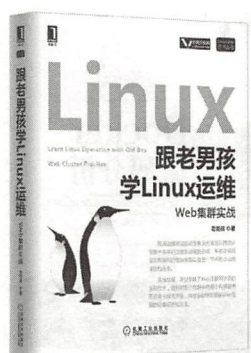
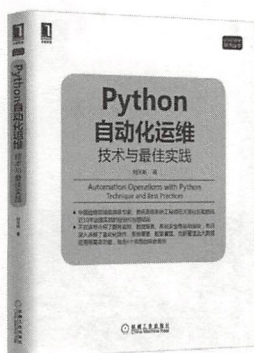
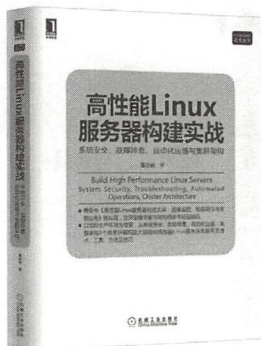
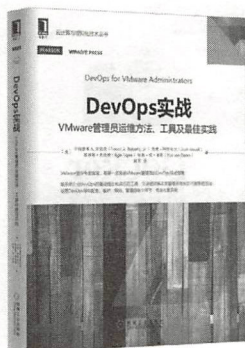
## 华章容器技术经典





# 推荐阅读

## 运维工程师进阶必读



## 内容简介

网络运维自动化资深专家撰写，8位专家联袂推荐，网络工程师转型必备指南。以场景与实践驱动，涵盖NetDevOps理念、常用工具、编程基础、网络运维常用Python模块与网络设备的数据处理等，注重实用性与友好性，全书分为5篇，共计14章内容。

**概念篇**（第1~2章），阐述NetDevOps是什么、怎么做、技术框架，使读者能清晰了解NetDevOps能给他们带来什么，从何入手，如何开展NetDevOps工作。

**基础篇**（第3~6章），介绍如何构建NetDevOps的工作环境以及在这些环境中的常用工具，提高读者日常维护工作的效率与准确度。

**提高篇**（第7~9章），讲解Linux环境编程、Python脚本编程、常用数据结构，学习开发一些在运维或者网络规划中能够使用的关键技能。

**实践篇**（第10~12章），采用案例的形式，带领读者掌握网络自动化运维、网络设计与规划中应对高频场景的技术与技巧，主要是网络数据的批处理，提高处理效率。

**案例篇**（第13~14章），通过3个典型案例来巩固提高NetDevOps相关知识和技能，更具体、更实用。





还在逐台配置、获取信息？

还在用文本比较工具准备变更方案？

还在用脑力计算路由表？

**网元和业务规模急剧膨胀，业务场景和数据多到力不从心！**

提高网络变更的效率；

提高操作网络设备的准确性；

缩小故障诊断的时间；

提高日常工作效率。

**那应该看这本书！**

NetDevOps是DevOps在网络领域的实践，通过这种全新的方式，填补开发不懂底层网络，运维人员自动化研发水平低，从而解决难以沟通，难以重构、优化网络的难题。



投稿热线: (010) 88379604  
客服热线: (010) 88379426 88361066  
购书热线: (010) 68326294 88379649 68995259

华章网站: [www.hzbook.com](http://www.hzbook.com)  
网上购书: [www.china-pub.com](http://www.china-pub.com)  
数字阅读: [www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导: 计算机\网络

ISBN 978-7-111-59909-8



9 787111 599098

定价: 79.00元